# Trustversion: A Secrecy-protected Version Control System

**Marta Isabel Ribeiro Sequeira**

Thesis to obtain the Master of Science Degree in

## Information Systems and Computer Engineering

Supervisor: Prof. Nuno Miguel Carvalho dos Santos

### Examination Committee

| | |
|---|---|
| Chairperson: | Prof. José Carlos Alves Pereira Monteiro |
| Supervisor: | Prof. Nuno Miguel Carvalho dos Santos |
| Members of the Committee: | Prof. Pedro Miguel dos Santos Alves Madeira Adão |

**October 2016**

# Acknowledgements

First and foremost, I would like to deeply thank my advisor, Prof. Nuno Santos, for the huge guidance and support and even bigger patience during the development of this thesis. Without his help, his knowledge and his words of encouragement, I could not have done this.

I would also like to thank my parents for the unconditional support throughout all my life, and my friends and colleagues, specially the ones that have already finished their Master's Degrees, for all the advice and experiences they shared with me.

Lastly, I would like to thank André Aparício for his interest and his ideas that contributed to the development of this work, but mostly for his patience, emotional support and giving me the motivation that kept me going.

Thank you.

Lisboa, October 2016
Marta Isabel Ribeiro Sequeira

For my parents,

# Resumo

Sistemas de Controlo de Versões são vastamente utilizados por programadores para guardar os seus projetos de *software* em locais remotos e para manter um histórico de todas as modificações nos seus ficheiros. Considerando que os projetos de *software* podem constituir informação privada para os programadores, colocar tais projetos em servidores remotos pode levantar questões de segurança, uma vez que os servidores podem não ser confiáveis. Atualmente, os Sistemas de Controlo de Versões, em particular o Subversion, não fornecem confidencialidade de dados face a servidores comprometidos. Este projeto tem como objetivo proteger estes dados através de um sistema designado *Trustversion*, que fornece uma solução para realizar operações SVN mantendo os ficheiros dos programadores cifrados no repositório. Esta solução permite proteger os dados privados dos utilizadores, mesmo que estes recorram a serviços online de repositórios SVN, como o SourceForge, alojados em servidores remotos que podem não ser confiáveis. O Trustversion garante também um baixo *overhead* de utilização de armazenamento, aproveitando os mecanismos do Subversion para guardar dados eficientemente.

# Abstract

Version Control Systems (VCSes) are widely used by software developers to store their software projects in remote locations and to maintain a record of all modifications in their files. Considering that software projects may constitute valuable private data for developers, having such projects stored in remote servers can raise security concerns as the servers may not be trusted. Currently, VCSes, in particular Subversion, do not provide data confidentiality in the face of compromised servers. In this project, we designed a system to protect the data called Trustversion, which provides a solution for performing SVN operations while keeping the developers' files encrypted on the Subversion repository. This solution protects the user's private data, even if they use online SVN repository services, such as SourceForge, hosted in remote servers that may not be trusted. Trustversion also ensures low storage usage overhead by taking advantage of the Subversion mechanisms for storing data efficiently.

# Palavras Chave
# Keywords


## Palavras Chave

Sistema de controlo de versões

Privacidade de dados

Segurança

Servidor não confiável

Confidencialidade

Armazenamento

Dados cifrados

## Keywords

Version control system

Data privacy

Security

Untrusted server

Confidentiality

Storage

Encrypted data

# Contents

# List of Figures

# List of Tables

# Acronyms

**AES** Advanced Encryption Standard

**API** Application Programming Interface

**BDB** Berkeley DB

**CE** Convergent Encryption

**DBA** Database Administrator

**DBMS** Database Management System

**MLE** Message-Locked Encryption

**OS** Operating System

**PIR** Private Information Retrieval

**SFTP** SSH File Transfer Protocol

**SSH** Secure Shell

**SSHFS** SSH Filesystem

**UDF** User Defined Function

**VCS** Version Control System

# 1 Introduction

## 1.1 Motivation

Revision control has become an essential tool in project development for managing history of information and to allow multi-user contributions in documents. Examples of typical cases that benefit from revision control are software projects, LaTeX documents or system configuration files. Version Control Systems (VCS) implement this functionality by storing the files in another location usually remote, called repository, and providing methods for synchronization. There are two types of VCSes:

- *centralized*, which follows a typical client-server architecture. The users are VCS clients that access the information stored in the repository on the server. Popular centralized systems are CVS (1) and Subversion (2);
- *distributed*, which adopts a peer-to-peer architecture. In this case, every user's machine has a full copy of the repository and the synchronization is made with any peer containing the same repository. Popular distributed systems are Git (3) and Mercurial (4).

There is now a variety of web services that provide online storage repositories powered by VCS systems. Some examples of systems providing these services are: GitHub (5) supporting Subversion and Git, BitBucket (6) supporting Git and Mercurial, SourceForge (7) supporting Subversion, CVS, Git and Mercurial, and Google Code (8) and CodePlex (9), both supporting Subversion, Git and Mercurial.

Online repository hosting services are a convenient solution for users that do not possess the resources to create or manage remote repository servers. Furthermore, because these services guarantee high availability and some of them provide additional features to help with project management, such services are now widely used for storing revision controlled data, even for enterprise and personal usage. For example, SourceForge has currently over 430,000 software projects involving 3.7 million users (10) and GitHub has over 23.2 million projects involving 9.7 million users (11).

However, VCSes do not guaranty privacy of information on the server side. In fact, they are design to work solely with plaintext files. The only security guaranty these VCSes give is secure access control to files by users. This poses a threat to users entrusting private information to online servers susceptible to attacks such as intrusion or information disclosure. Let's imagine that a company uses a VCS hosting service to store a privacy-sensitive project that needs to be protected against the competition. The company will not be able to tell if the hosting server is compromised, either by an external attacker that was able to penetrate the server and has now access to all data stored in there, or a more simple internal attack by the service provider allowing information leakage. Even a curious or malicious system administrator constitutes a security concern.

Protecting the confidentiality of VCS files is not straightforward. The users or a simple system could try to protect the data by encrypting all the files before submitting them to the repository server, and decrypting upon receiving them from the server. However, VCSes employ mechanisms to avoid storing duplicated data which could be rendered ineffective. In fact, by "blindly" encrypting all file versions, VCS will not be able to detect which file parts remain immutable across versions and, therefore, it would store parts of the same data multiple times unnecessarily which would heavily increase the occupied storage on the server over time.

## 1.2 Goals and Requirements

This thesis aims to protect private data in public VCS services by securing the information on the repository through file encryption while allowing the VCS server-side operations to be performed over encrypted data. In order to achieve this main goal, our solution must fulfill the following set of requirements:

**Data confidentiality:** Guarantee data confidentiality by cyphering all documents in the repository.

**Storage space efficiency:** Consider storage space efficiency, since encrypted files can become much larger than plaintext files and the encryption schemes can interfere with the VCS compression algorithms. Minimizing the overhead in storage space is particularly important in VCS service providers because it can influence pricing plans and increase costs for the users.

**Client-side implementation:** The implementation of our solution must reside on the client side only. By avoiding alterations on the server-side code, our solution can be deployed independently of the server and could be used seamlessly with any existing online repository server.

To the best of our knowledge, there are currently no approaches that guarantee data confidentiality under this set of requirements. Recent work has shown to be possible to operate on encrypted data: to do arithmetic computations (12; 13), to search keywords (14; 15; 16; 17) and to sort the data (12). Some systems were implemented or complemented with a combination of these techniques to reach the same goal of operating over encrypted private data, namely Database Management Systems and Private Information Retrieval systems. However, the approaches taken by these systems cannot be applied to securing VCS services because the operations involved in a Version Control System are a lot different from retrieving information or performing SQL queries with a fixed structure. Another interesting technique is to encrypt the data using fully homomorphic encryption schemes (18), which allows multiple operations to be performed directly over the encrypted data. But these are asymmetric encryption schemes and might be too computationally expensive to use exclusively on encrypting an entire repository.

## 1.3 Contributions

This thesis presents Trustversion, a system to ensure data privacy for public Subversion repositories. Trustversion provides confidentiality of private data while ensuring storage space efficiency, as it

is coordinated with Subversion's mechanisms to reduce space usage. Our system runs exclusively at the client side and works with unmodified Subversion servers. An unmodified Subversion stores on the server only the changes made to the files and not every full version of the files. So the underlying idea of our solution is to encrypt those file changes that are sent to the server. Then we *trick* the server into believing that the user is just appending unrecognizable binary data to the files. The binary data is actually the encrypted file changes, but the server will be unaware of it. Because the encrypted information is being passed onto the server as append-only modifications on the files, Trustversion does not interfere with the normal behavior of Subversion. Moreover, by encrypting only the changes on the files, the encrypted information will be much more compact then a simple approach of encrypting the entire files. We call this method *append-only encryption*.

To design Trustversion, we will address the following main challenges. First, the decision about the most adequate cryptographic schemes to support our approach. These, alongside the structure of the system and its operations, will determine the schemes to use when ciphering data and operating on top of it. Second, Trustversion must contemplate how much performance and storage space overhead can be introduced by the cryptographic schemes used supporting the SVN operations on the encrypted data. The overhead must be reduced.

As mentioned above, the implementation of Trustversion was made for Apache's Version Control System: Subversion (2). We implemented an adaptation of Subversion supporting basic encryption of the data, simulating the approach of manually ciphering the data before sending it to the server. We also implemented another adaption of Subversion supporting append-only encryption, our solution that takes into account the Subversion's mechanisms for saving storage space by keeping file changes as encrypted appends on the original file. Both these systems were compared side-by-side to analyze the impact of interfering with Subversion's server mechanisms to efficiently occupy disk usage.

The evaluation was performed on three different systems: the Subversion system, our implementation of a modified Subversion supporting basic encryption functionality and our solution, Subversion with append-only encryption. We applied these systems to four big software projects: CloudStack (19), CouchDB (20), Perl (21) and Ant (22). We evaluated the results of these projects using the three systems in terms of storage space occupation and execution time performance, and compared them in order to see the impact of each approach and the overhead they introduced.

To obtain the repositories to use as test cases and to be able to apply all three systems on them, we developed automated scripts to retrieve repositories from publicly available servers and to copy each repository by reproducing every revision in the repository and, thus, simulating the activity history of the original repository.

## 1.4 Structure of the Document

In this document, we will start with some background in Chapter 2 describing how VCSes work, in particular, the internal operations of Subversion. Seeing that low-level documentation for Subversion is very scarce, the information about the internal behavior was mostly achieved studying Subversion's source code. Such knowledge was crucial to design our solution, and thus, it is the basis to understand

how Trustversion will operate. Chapter 3 provides an overview of existing work approaching the same type of problems we face or providing techniques that can be useful for our purpose of securing private data and operating over it. The architecture design and implementation of Trustversion, including a basic encryption approach and an append-only approach, are laid out in Chapter 4. The details of how Trustversion was evaluated and the results we obtained are shown and discussed in Chapter 5. We finish this thesis with some conclusions and ideas for future work in Chapter 6.

# 2 Background

Version Control Systems (VCS) were designed with the goal of keeping information about every change made in files, and having the ability of rolling back changes. The most popular VCSes are CVS (1), Subversion (2), Git (3) and Mercurial (4). In this chapter we give an overview of how VCS work, followed by a more in depth explanation of some internal mechanisms of Subversion.

## 2.1 How VCSes Work

The main application of VCSes has always been revision control of software projects. Revision control is the concept of registering changes in files with an associated revision number, which identifies each set of changes at a given point in time. In the context of software projects, revision control allows users to trace when a certain source code modification was performed and by whom. With Subversion, for example, every project starts at revision 0, and for each set of changes submitted, the revision number is incremented.

All projects under a VCS include at least one repository (some VCSes support multiple repositories per project). A repository represents the root directory of the project, i.e. the tree of files that are part of the project. The users participating in the same project use its repository (or one of the repositories) to synchronize changes and revisions. The synchronization operations are always issued by the users, so all VCSes provide similar commands to achieve this synchronization, the most relevant being:

**checkout** (or "clone" in Git and Mercurial) is the operation of copying a repository to the user's local filesystem and it is the first step to participate on an already existing project. This operation creates a local copy of the repository in which the user is supposed to make the file modifications, and is called the *working directory* or *working copy*.

**commit** ("commit" and "push" in Git and Mercurial) is the command to associate a new revision number with the current set of file changes since the last revision and submit them to the repository.

**update** ("pull" in Git and Mercurial) is the command to pull all changes submitted to the repository that have not yet been applied to the user's working copy, thus updating the user's files with those changes. If a file has been modified locally when an update is issued and that file needs to be updated, then a merge operation is automatically initiated to insert the changes on the file without interfering with the local modifications. If the merge is unsuccessful, then the file enters a state of conflict that has to be manually resolved before the next commit.

**add/remove** to schedule files to be added or removed from the repository in the next commit operation.

**status** prints the current state of the working directory and its files, namely if there are changes to be submitted or if there are files with conflicts to resolve.

**diff** is a command to print differences on files between two specified revisions by comparing the contents of the files of both revisions and printing only what differs.

**log** shows the history of revisions and the logs associated with them. These logs are messages requested to the user when performing a commit, to be appended to the revision.

These operations are common to all VCSes. Each VCS then provides additional operations to support functionalities either specific of that VCS or to their architecture and way of implementing the revision control.

## 2.2 The Subversion Internals

This work focus on the Apache's VCS: Subversion (2). Subversion has been the most popular centralized VCS for years, being currently used in big software projects such as Apache's web server (23), GCC (24), Mono (25) and FreeBSD (26).

Starting from the architecture at a high level, Subversion follows a client-server model that includes one Subversion server, at least one Subversion client and libraries implementing the VCS operations to be used by the server and the clients. Figure 2.1 gives a top level view of the architecture. The client interacts exclusively with the *Client Library* through an API, which was designed to allow for various implementations of a Subversion client. Those implementations exist besides the default command-line client Subversion already provides, and one example is a widely used graphical Windows client called TortoiseSVN (27).

Both Subversion's server and the *Repository Access Library* manage repositories, located at the *Filesystem Layer*, and both the *Client Library* and *Working Copy Library* manage the working copy and all synchronization operations on the client. We can also see that Subversion supports two backends for storing information locally: the FSFS, the Subversion's filesystem built on top of the OS filesystem, and the Berkeley DB (BDB). They offer different performances for different SVN operations, although FSFS was developed as an improvement of BDB and is the default backend for Subversion since version 1.2 released in 2005 (28). BDB has been deprecated since the current version of Subversion – 1.8 – released in 2013 (29).

To understand how Subversion optimizes storage space usage, the internal behavior of the user operations on Subversion must be explained. This knowledge is essential to better understand the problem we want to solve in this thesis. Let us imagine a scenario where we want to commit a changed file and a modified file in the following directory tree:

```
/
M  /file1.txt
/foo/
A  /foo/file3.txt
```

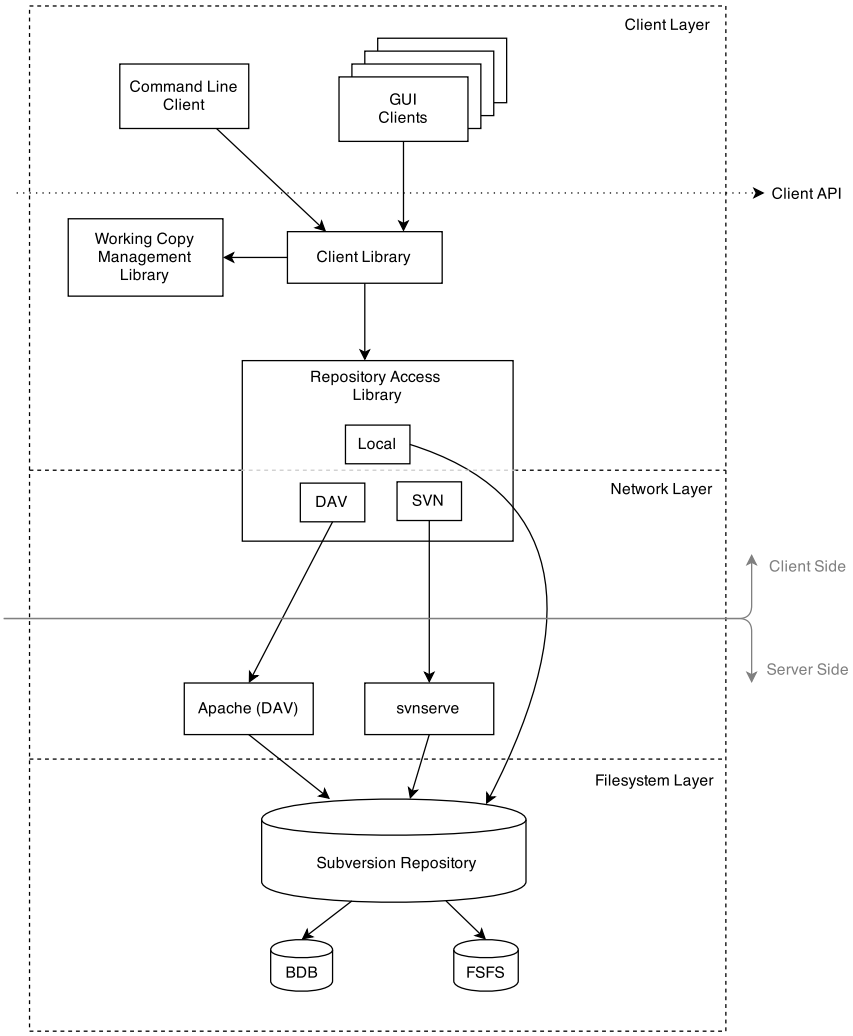


Figure 2.1: Subversion Architecture Design

Given that “M” stands for “modified” and “A” for “added”, this is the state of the working copy before committing. Figure 2.2 shows how a Subversion commit operation works in this scenario, using a sequence diagram.

As we can see in the commit diagram, the changes in our scenario are transmitted through a sequence of function calls. First the root directory (“/”) is opened through the ”Open Root Directory” function call, then the modified file “file1.txt” is opened (to be changed) through the “Open File” call. Recursively along the directory tree, for every file to be added or modified, all the directories on their path need to be opened. For this reason, the next step is for the directory “/foo/” to be opened, followed by the function call to add the file “file3.txt”. After all the directories and files are opened or added, recursively up the directory tree, the changes are actually transmitted through the “Apply Text Delta” function call, and the directories and files are closed with the “Close File”/“Close Directory” calls. All these function calls are applied by the client on an object called *Editor* that was initially requested to the

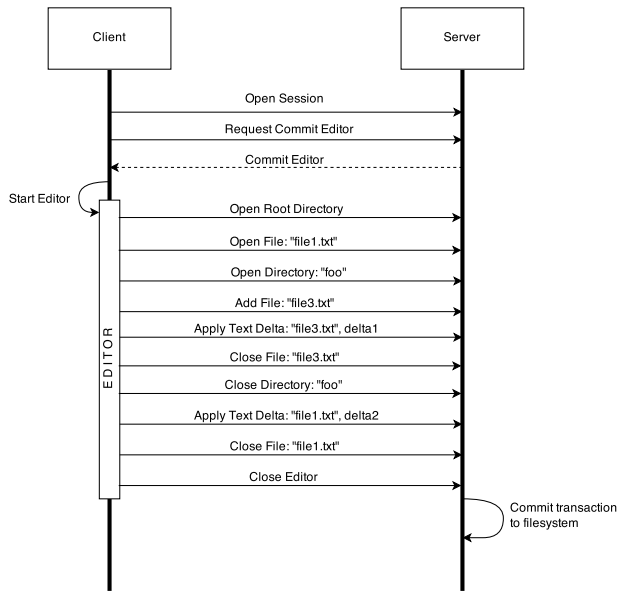


Figure 2.2: Sequence diagram of a commit operation when /file1.txt is modified and file /foo/file3.txt is added.

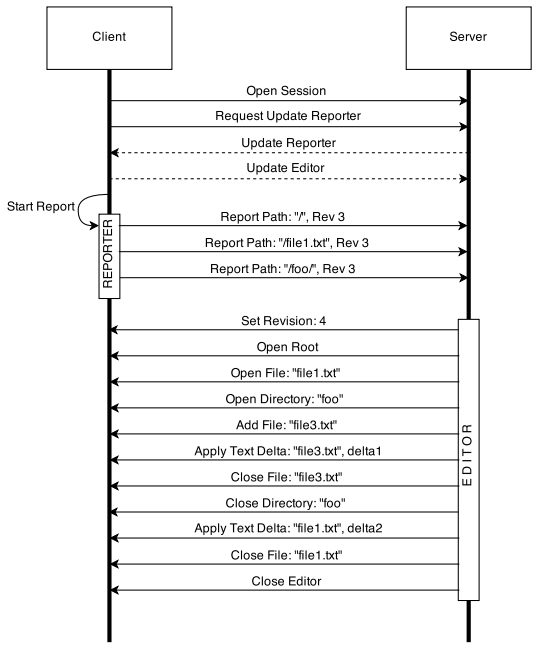


Figure 2.3: Sequence diagram of an update operation when /file1.txt is modified and file /foo/file3.txt is added.

server, as it can also be seen in the diagram of Figure 2.2.

An **Editor** is used in most Subversion operations for communication between clients and server. It has two purposes: for a client to edit the directory tree stored in the server's repository (e.g. with a *commit* operation), or for the server to edit the directory tree of a client's working copy (e.g. with an *update* operation). An editor is an interface declaring a set of functions that each SVN operation implements differently. Some of them we have seen in the scenario above, but all the main editor functions are the following:

- *Delete Entry*: for deleting a directory or file;
- *Add Directory/File*: for adding a new directory or file;
- *Open Directory/File*: for opening a directory or file to make changes inside it;
- *Change Directory/File Property*: for altering the properties of a directory or file (for example, permissions or ownership of the directory or file);
- *Apply Text Delta*: for altering the contents of a file;
- *Close Directory/File*: for closing a directory or file after all changes have been made;
- *Close/Abort Edit*: to finish or abort (in case of error) the execution of this editor.

In the particular case of the commit operation, the editor instance to use is a commit editor implemented by the server, and its functions, working as stubs, are called by the client sequentially as a

way to reproduce in the server the changes introduced since last revision. The scenario above and the diagram in Figure 2.2 showed how a client reproduced the local changes on the server. This is the way a client transmits modifications in the working directory to the server.

Let's imagine now trying to update another user's out-of-date working copy to the revision generated by the commit in the scenario above. Assuming the repository is at revision 4 and all files in the working copy are at revision 3, the current state of the working copy is an out-of-date `/file1.txt` and a missing `/foo/file3.txt`. Therefore, updating the working copy will put the directory tree in the same state as the commit operation above:

```
/
M  /file1.txt
/foo/
A  /foo/file3.txt
```

This time "M" indicates that the file will be "modified" and "A" that it will be "added". Figure 2.3 shows a sequence diagram of the behavior of an update operation in this scenario.

Notice the similarities between the commit and update diagrams. The same editor functions are being called, because the same changes are being applied, but in the update diagram the calls are made in the opposite direction, from the server to the client. The editor in this operation is an update editor implemented by the client, that was sent to the server in the first interactions present in the diagram. There are also additional calls ("Report Path") before the editor calls during the update operation. The client reports the current state of its working directory to the server using the "Report Path" function calls, which provide information about the revision of each path in the directory tree. In this scenario, the client is reporting each path in the working directory individually as being in revision 3 (different paths may be at different revisions). Notice that the path `/foo/file3.txt` is not being reported, because it does not exist in the client's working copy before the update. The report process provides the necessary information for the server to know which changes to apply on the client through the editor. The report process is performed by the client using a *Reporter* requested to the server.

A **Reporter** is how the server or clients report the current state of their repository or working copy, respectively. Much like the *Editor*, it is used by most SVN operations involving communication between the server and a client. It is also an interface with a set of functions whose implementation is decided by the SVN operation. However, the reporter is simpler because it has a much smaller set of functions, used only for informing which files are present in the repository/working copy and which revisions they are in.

In Figure 2.3 we see an update reporter, an instance of the reporter interface for the update operation, implemented by the server. It is requested by the client to report its state to the server using the reporter's functions.

Commands like *status* and *diff* behave in a way very similar to the update operation if the user specifies to also use information from a repository. They also use a reporter to provide information about the working copy and an editor to transmit changes. Although, unlike the update operation, these commands implement the editor such that no modifications are performed on the files. The server calls the same editor functions as for an update, but they do not actually alter the state of the directory tree

and rather just print information about the changes. The *status* is for printing a description of changes in the directory tree, and *diff* is for printing differences in file contents.

The focus is on the commit and update commands because all other most commonly used Subversion commands perform the same overall procedures.

### 2.2.1 Managing File Content Differences

We've seen how changes in the tree of files are managed, in particular how to transmit changes in the content of a specific file. These changes can be sent through the network. They are also intended to save storage space on the server. Thus, only the sections of the content that were modified are sent instead of the entire file. Subversion computes which parts of the file differ towards the version from last revision and generates a *delta* for that file. A delta is a description, with a compressed format, of the modifications introduced in a file and is generated by Subversion's diffing algorithm (30; 31). This is why the operation of altering files is called *Apply Text Delta*.

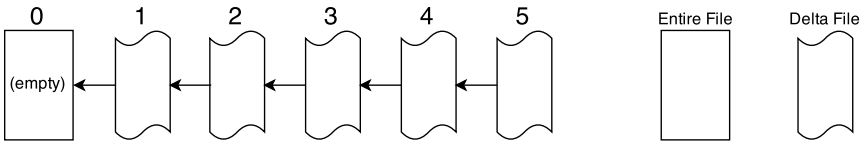


Figure 2.4: Representation of how deltas are stored in the FSFS backend. The numbers represent the revisions on the file.

Deltas are how the server can store every version of every file in an efficient manner. In the case of the FSFS backend, the changes on a file in each revision are stored as a delta towards a previous revision of that file, as seen in figure 2.4. The revision 0 is always an empty revision and the first commit of a file is stored on the server as a delta towards revision 0 (which translates to the entire file).

If we were to encrypt the entire files before committing them, the purpose of the deltas would be irrelevant. Performing a diff between two versions of the same file encrypted separately, produces two very distinct binary files, even if the plaintext versions are almost equivalent. A delta computed for these two versions of the file would probably be almost as big as the file itself, so it would not optimize storage space usage at all. Over time, the increase in storage occupied in the server would be proportional to the files sizes times the number of deltas for each file.

### 2.2.2 Managing Duplicated Information

Besides the delta mechanism described in the section 2.2.1 above, Subversion also has a mechanism to avoid storing duplicated information, called *Representation Sharing*. This mechanism optimizes storage space regarding *branches* and *tags*.

Both *branches* and *tags* are common concepts in VCSes. A *Branch* is a copy of the repository's current state which allows to perform modifications to the repository's files in parallel with possible mod-

ifications on the original files. This is typically used in software projects to develop new (possibly experimental) features without interfering with current developments on the same project. All the modifications applied in a branch can then easily be discarded or merged with the current state of the original files (called *trunk* in Subversion). A *Tag* is a representation of a repository at a specific point in time. A tag saves the state of the repository in a certain revision and can add a meaningful name to it. In software projects, typically, each tag denotes a version of that software (e.g. ”v1.2.1”).

In the case of Subversion, the trunk, branches and tags are all simple directories in a repository. Each branch or tag is a complete copy of the tree of files and directories inside the trunk directory. This means that every time a branch is created, all the content in the trunk directory is duplicated. Thus, in order to optimize storage space, Subversion keeps only one representation of the duplicated information on the server. If multiple unmodified copies of a file exist across the entire repository, only one file is kept on the server.

Given that software projects widely use branches and tags, this can reduce a lot of overhead for big software projects.

## Summary

These are just some of Subversion’s internal functions and techniques and a small overview of how they are implemented. Subversion is a much more complex system than what we presented in this chapter. Still, we gave a deeper understanding of the internal behavior that was relevant knowledge to reach our solution and also provided a brief introduction to how overall VCSes work. In the next chapter we go through the current work that proposes solutions for the general problem of securing user’s data.

# 3 Related Work

We present current existing work among different areas that, similarly to Trustversion, address the general problem of securing privacy-sensitive user data and computations from a untrusted server. We will also see why these systems do not satisfy our requirements in securing remote VCS systems. This chapter describes the related work in different scopes of data protection, such as protecting data on VCS repositories (Section 3.1), on databases (Section 3.2), for keyword searches (Section 3.3) and on systems with deduplication mechanisms (Section 3.4). Each subsection describes representative systems where such techniques are implemented in practice.

## 3.1 Protection of Version Control Systems

This section describes two proposed solutions for protecting data under revision control in a Version Control System (VCS), which is the main goal of Trustversion as well. We explain these two schemes below and we see why they do not solve all requirements set for Trustversion.

### 3.1.1 Encrypt the Repository

Encrypting the repository on a filesystem level is a possible solution for ensuring data privacy and it works for any VCS. The idea is to store the repository in an cryptographic filesystem, like EncFS (32), and mount it remotely, as shown in Figure 3.1.

Files stored in EncFS are encrypted transparently while the user manipulates them normally at the client side. After specifying a directory to be encrypted, EncFS maintains the files on that directory unencrypted and mounts a separate directory with the encrypted files. In this case, the repository consists either of the directory to be encrypted or a directory already present in the local filesystem. But it is highly unlikely that a regular user is able to create an EncFS filesystem on any remote server, due to lack of administrator permissions, so the filesystem has to be created on the user machine.

Thus, to have a remote repository, the user could mount on the server its EncFS encrypted directory using a remote filesystem service such as SSHFS (33). SSHFS provides a way to mount directories on remote servers through the SSH File Transfer Protocol (SFTP). But, in order for this approach to work, the server must allow users to use the SSHFS service, which may not be available in a typical repository server. GitHub (5) is an example where this approach would not be possible because it does not make the SSHFS service available to the users.

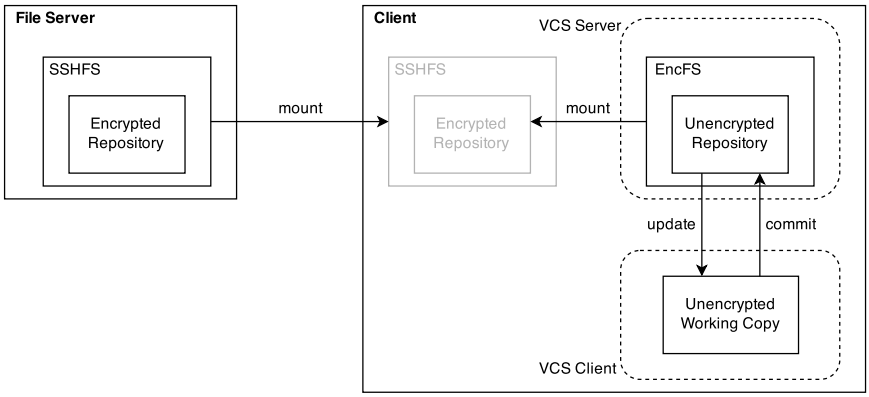


Figure 3.1: This diagram shows a repository encrypted with EncFS stored remotely in the server through the SSHFS. The encrypted repository is used on the user machine as a regular directory, but it is actually stored remotely, thus the gray boxes. We can also see that both the VCS client and server are running on the user machine and work with plaintext files.

This solution may also not behave well with multiple users using concurrently the same repository. Each user machine will be running a separate instance of the VCS server, although in practice all the instances will be working on the same files physically. The behavior in this case is unpredictable.

The following approach proposes a much simpler and cleaner solution to the user.

### 3.1.2 Crypto Hook

Crypto Hook (34) is a TortoiseSVN (27) plugin. TortoiseSVN is a graphical client implementation for Subversion. The main idea of Crypto Hook is to intercept the commit operation and encrypt all the files before they are committed to the server, and intercept the update operation to decrypt all files after the modifications are applied.

This plugin works by having User Defined Functions assigned to certain phases of the commit and update operations, running as hooks on the client. The function for the pre-commit phase takes a list of files to be committed and encrypts every file and its corresponding *svn-base file* (which is the version of the file from its last revision), then the commit proceeds. On the post-commit phase, the function takes the previously encrypted files and deciphers them, and their corresponding svn-base file, so the user can continue working on them. As for the update operation, in the start-update phase, the respective function encrypts every file and its corresponding svn-base in the working directory, since it does not know which files will be updated. The update proceeds normally, merging encrypted files from the repository to the out-of-date encrypted files in the working copy. After that, in the post-update phase, the function deciphers every file and possible new files brought from the repository, and their corresponding svn-base file.

The reason why it needs to cipher all the files on update is because Subversion itself will check for local file modifications before starting the update, by performing a simple diff against its version on the repository. Even if the file is not out-of-date, it will be in plaintext in the working copy and will not match

the encrypted version on the repository unless the plugin re-ciphers it again before the comparison.

Since, on each revision, SVN only stores on the repository the parts of the file that were modified and not the entire file (for space efficiency), for this plugin to work, we have to start from an empty repository. It cannot be added to an existing repository already containing plaintext files, unless the project on that repository is moved to a new, empty repository.

This plugin supports multiple users accessing the repository, by using a secret key to cipher the files that must be shared among all users of the repository. The problem of key distribution, however, falls out of the scope of this system.

Even though this plugin introduces a better solution than encrypting the repository at the filesystem level, it still has a few problems. Besides the system only working with a specific client implementation of the Subversion VCS, it does not take any advantage of the Subversion mechanism for storage efficiency.

### 3.1.3 Discussion

Both these schemes, Encrypted Repository and Crypto Hook, have important limitations. On the first scheme, all clients use the same repository on disk but with different instances of the Subversion server running on the machine of each user. This introduces some unpredictable behavior. It is unknown how this scheme would behave in the face of concurrency, because Subversion servers are not prepared to run this way. It may possibly cause file inconsistencies. With multiple instances of the server managing the same repository, there could even be different versions of the server software among the instances. Thus it is too unstable to be a desirable solution. It is also not compatible with existing VCS services, such as GitHub or BitBucket.

The second scheme is less complex for the users and can give guarantees of data confidentiality, although it does not take advantage of the Subversion mechanisms to efficiently store files, as described earlier. A Subversion repository only stores diffs representing the changes in the files in each revision, but a diff of an encrypted file in relation to its old version will be the new entire file, which will eventually generate a big storage expansion for that repository. This is also not a desirable solution, taking into account that there are very large projects under revision control.

## 3.2 Protection of Database Management Systems

There are different areas in which the kind of operations performed on data allow for those operations to be adapted to work with encrypted data. In the Version Control Systems area, we saw that there are currently no systems that protect the user's data satisfying our requirements. In this section, we talk about the Database Management Systems area and what solutions have been developed to operate over encrypted data inside databases. In particular, we discuss how to maintain an entire database encrypted, but still allowing data retrieval and modification without the server ever knowing the contents of the database, and without interfering with the normal operations of a DBMS. This work relates to Trustversion in the sense that they both intend to maintain all data encrypted in the server, allow for the underlying system to work properly and allow for intelligently manipulate data already ciphered.

### 3.2.1 CryptDB

CryptDB is a system that aims to provide full confidentiality of data inside a database by performing SQL queries seamlessly over the encrypted data. CryptDB (12) is composed of a proxy to the database and some User Defined Functions (UDFs) applied on an unmodified DataBase Management System (DBMS). It is assumed that the DBMS can not be trusted and can be maintained by a malicious DataBase Administrator (DBA). The main purpose of the proxy is to intercept and rewrite the user queries to execute them on the encrypted data, and also to encrypt and decrypt the data in the queries and in the results. The proxy defines what keys are used in encrypting the data on the database.

The challenge of performing queries over encrypted data is that the columns of database tables have multiple possible types of data with multiple different purposes. Columns for strings of characters may be used for equality checks (in `WHERE` clauses) while columns with numerical values may be used to perform calculations (like `SUM` operations). A single encryption scheme cannot satisfy all possible ways of querying columns, so the most appropriate encryption scheme should be chosen for each column depending on its purpose. Furthermore, a single column may be selected for multiple purposes, so it should use multiple encryption schemes. But how?

Every column of every table has an onion of encryptions, meaning that it is encrypted several times and added in layers. Each layer uses a different kind of encryption, so CryptDB resorted to various types of encryptions featuring different trade-offs between security and functionality:

- random - very strong ciphering that uses always a random IV;
- deterministic - allows for equality comparisons;
- order-preserving encryption - allows for order comparisons (value $>$2);
- homomorphic encryption - allows for `SUM` operations;
- join and ope-join - an extension to “deterministic” and “order-preserving”, respectively, to allow join expressions;
- word search - to support `LIKE` expressions.

The onion layers are added with a particular order: the layer whose encryption method reveals more information (for example *order-preserving*) is the layer that will be added first and so on, so that the layer on the surface of the onion will be the least revealing (for example *random*). Then, these layers are stripped depending on the operations in the queries to be applied to that column.

CryptDB also solves the problem of having multiple users accessing the same data by chaining encryption keys to user passwords. The system resorts to principals, each one having a key associated. Principals can be users, groups or other entities, and the way the chaining works is: if principal A speaks for principal B or has access to principal B, then principal B’s key is encrypted with principal A’s key. Users’ keys are their own passwords. For example, if Alice and Bob have access to message with id 5 on the database, then message’s key will be encrypted with Alice’s key, and separately with Bob’s key and both are stored.

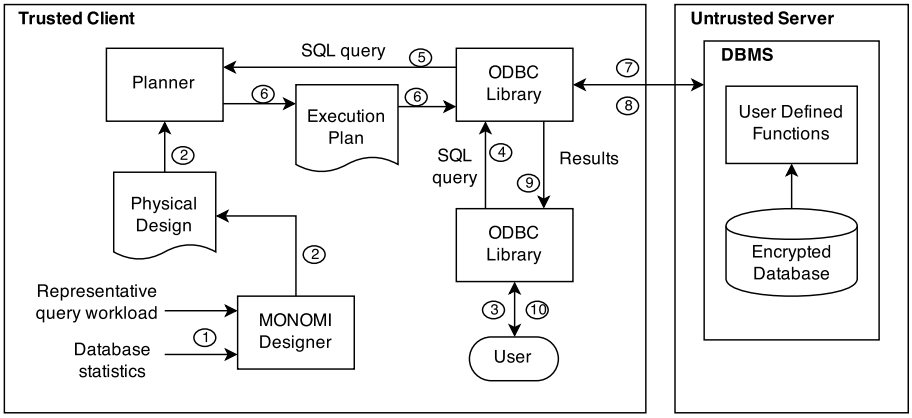


Figure 3.2: Architecture of the MONOMI system.

This approach of chaining keys may be a little inefficient for storage. For each user that has access to a data entry on the database, the key used to encrypt the data will be encrypted with the user's key. Every key encrypted with another key is added as an entry in the database. If we disregard the possibility of existing a chain of principals, the worst case scenario is that all users have access to all the data. In this case, the total number of entries with encrypted keys that are added to the database is the number of users times the number of data entries in the database. By taking the chain of principals into account, we also have to multiply by the number of principals in each hierarchy.

### 3.2.2 MONOMI

MONOMI (13) presents a solution for executing analytical queries on an encrypted database, since CryptDB does not deal well with this type of queries. The approach taken in this system is to split the computation between the server and the client: MONOMI tries to process the data in the server as much as possible, leaving only computation to the client whenever the computation on the encrypted data is not possible or not efficient enough in the server.

The way MONOMI stores the encrypted data and processes it is based on the solution proposed in CryptDB. Each column is encrypted with the ciphering method most adequate to the queries executed on it although, unlike CryptDB, it is only encrypted once, there are no onion layers. This solution, however, imposes the same limits as CryptDB does on analytical queries which justifies the addition of client-side computation. In MONOMI, only the part of the query that the server is able to compute is sent to it encrypted, and the results produced by the server are then decrypted and sent to the client along with the remaining of the query for the client to process. An example query representing this situation is:

```
SELECT SUM(price) AS total
FROM orders
GROUP BY order_id
HAVING total > 100
```

This query shows only the orders with a total price superior to 100. It fetches all rows from the `orders` table, groups the orders by the `order_id` and sums all the prices for each `order_id`. Then only the rows whose total sum of the prices are above 100 are shown.

The `HAVING` expression cannot be processed by the server since the `total` column must be using homomorphic encryption. That is the necessary encryption scheme to support a `SUM` operation, but, at the same time, homomorphic encryption cannot be applied to order comparisons. In this case, the query can be split. The query without the `HAVING` expression will be processed by the server and its results returned to the client. Then the client will use the results from the server to process the remaining `HAVING` expression and provide the final results of the entire query.

One of MONOMI's characteristics is its complex architecture. Even though the system inherits CryptDB's approach, their solution requires previous knowledge about what kind of queries will be executed so that it can decide the best encryption for the data. The architecture to support this is represented in figure 3.2 and includes:

**A designer** that takes as input a subset of the queries or a workload representing the query set to be executed and statistics on the data to be inserted in the database, and produces a *physical design* for the untrusted server, containing the encryption schemes to use on the columns.

**A planner** that produces an execution plan defining how to split the queries and which part is executed by the server or the client.

**An ODBC library** used to issue normal queries on the database. It is also the only component that has access to the keys.

**User Defined Functions** on the database to support certain operations on the encrypted data

MONOMI has also one other problem: sending a large number of database records to the client whenever queries are split. This introduces a lot of inefficiency when dealing with more interesting analytical queries that retrieve huge amounts of data.

### 3.2.3 Discussion

Neither of these systems is suitable for ensuring data privacy for VCS services, because DBMSes have the advantage of working with small pieces of data mostly independent from each other. Each cell of a database table is a piece of data that is encrypted separately and each column can be using a different encryption scheme. In a VCS, we also want the system to be able to manipulate parts of the encrypted data (files in this case) in a repository, but files are large amounts of contiguous encrypted data so our solution cannot be achievable in the same way.

For example, these systems can determine one column will use deterministic encryption for fast equality checks and another column will use homomorphic encryption for performing mathematical operations over the encrypted data. But a VCS does not contain subsets of data with different purposes as databases do. It contains only files that share the same purpose of managing their changes. And managing changes may need both equality checks and computing over encrypted data. We could try the

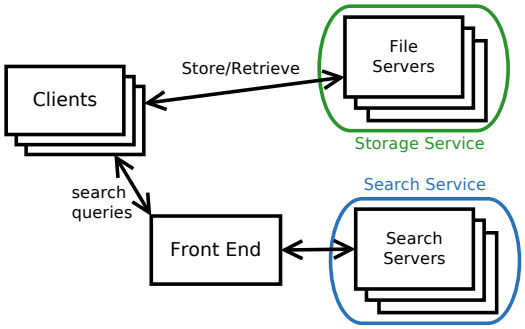


Figure 3.3: Architecture of the Mafdet system.

onion encryption layers approach, but it would not be the best method to use in VCSes, as encrypting entire files multiple times would introduce a huge performance overhead.

Nevertheless, CryptDB and MONOMI introduce interesting ideas for dealing with the encrypted data such as the use of a proxy or the split execution between client and server.

## 3.3 Private Information Retrieval

Private Information Retrieval (PIR) is also an area that allows for data to be encrypted and has some solutions covering the problem of operating on that data. More specifically, this area of research is concerned with searching keywords on encrypted documents in order to retrieve only the documents of interest (i.e. that match the keyword). PIR aims to solve the problem of protecting textual documents on untrusted file servers while allowing keyword searching on them and retrieval of those same documents based on the search results.

### 3.3.1 Mafdet

Mafdet (14) is a system to perform keyword search on encrypted data located on a distributed storage system. Mafdet's architecture provides two separate services — a search service and a document storage service — and has three main components as represented in Figure 3.3:

**File Servers:** typical file servers with the encrypted documents to be searched over, supporting any file server architecture.

**Search Servers:** servers that store metadata for each document on the file servers. The search servers take only keyed hashes of the keywords and the metadata stores information about whether a (hashed) keyword is present in the document. So they can locate documents containing the searched keywords, and also cache previous search results.

**Front End:** Functions as a load balancer and failover, and keeps a map between the documents and the search servers containing the documents' metadata. It is responsible from managing the files (add, remove, etc) and for issuing the searches, i.e. it sends the searched keywords, hashed with the client's key, to the search servers for them to perform the search.

The clients encrypt the documents and generate a bloom filter for each document to add to the search servers through the front end. The documents are stored in the file servers. Afterwards they can perform search queries using the Front End. When a client wants to perform a search, it uses a keyed hash function to obtain hashes of the keywords to search for, using a key that is private to the client. This prevents the server from learning the search keywords and also to perform itself searches of arbitrary keywords. The search query with the hashed keywords is issued to the Front End. The Front End redirects the search query to the search servers, and they use the documents' metadata to assess which documents may possess the given keywords. The result is a list of documents that is sent back to the Front End and, since it has a map between the documents and the file servers, it will retrieve all the documents in the result list. The set of documents is then sent back to the client.

Mafdet identified a challenge with indexing documents using keywords. An index of keywords pointing to the documents where they are present can reveal some information even if the keywords in the index are encrypted. This is due to the addition or removal of documents, which would issued an update of the index in the positions of the keywords present in the document being added/removed. This process would allow the server to infer some correlations between documents sharing the same keywords.

The above challenge motivated Mafdet to be mostly based on Bloom (35) filters, an algorithm to determine if a member is in a set with some probability using efficient data structures and hash functions to fill those structures. Bloom filters can determine that a member is not in the set with 100% certainty, but can only determine if a member is in the set with some probability. Every document has a different bloom filter associated and uses different hash functions so that their information cannot be correlated. The members in the bloom filters are not the keywords of the document but rather keyed hashes of the keywords, using the client's key. These filters are the metadata stored in the search servers associated with each document.

Appended to the keywords stored in the bloom filters is the number of the chunk in the document where the keywords occur, giving some more detail on the location of the keywords. This is useful for getting a more approximate number of occurrences of a keyword in a document, or the proximity between two different keywords. The following criteria established by Mafdet uses that information to perform ranking of search results by relevance to the search query:

**Query word proximity** Favors documents that contain multiple keywords from the same query on close proximity in the document, such as occurrence of two keywords in the same chunk.

**Occurrence count** Favors documents that contain the most number of occurrences of the searched keywords.

**Similarity** Favors the occurrence of uncommon words over common words.

**Distinct query keyword count** Favors the occurrence of distinct keywords from the query over the occurrence of a single keyword multiple times.

Given that the documents are encrypted with the client's key and the searches are performed with encrypted keywords, this system provides guarantees that the server can never obtain plaintext data. It also helps limiting statistical attacks since the server itself cannot perform searches with arbitrary keywords (the client's key is needed). However, the server can still observe the results for search queries performed by the clients.

The adoption of Bloom filters as an alternative to indexing the documents with their keywords also limits statistical attacks because there is only one bloom filter per document, which is created by the client when a document is added to the servers and it never needs to be updated. Besides, as opposed to indexes, with Bloom filters you can only check if a certain keyword is present in the document if you have the encrypted keyword. With Bloom filters, it is not possible to retrieve which keywords are present on that document. Additionally, this system assumes the attacker is passive, can eavesdrop but cannot compromise the integrity of the data or the results.

For performance, they cache results from previous searches and cache frequently used bloom filters.

### 3.3.2 Mylar

Mylar (15) intends to protect web applications' data from untrusted web servers by providing a way for the web applications to work with encrypted data. The main goal of this system is to protect only the data stored by the web applications on the untrusted web servers. The computations Mylar supports over the encrypted data is solely encrypted keyword search. It also provides detection of tampering with client-side code (such as JavaScript).

Mylar allows for the application and the servers to be compromised and still guarantee data confidentiality, since all data marked sensitive is encrypted and decrypted only on the client and the server never has access to the unencrypted data. The client's code can be checked for integrity before having access to the user's key to guarantee the key or the plaintext data cannot be obtained if an attacker has modified that code. The server keeps the encrypted data and Mylar provides the support on the server to perform encrypted keyword search over the encrypted data.

Mylar's architecture consists of:

- a client-side library to encrypt and decrypt data the client stores in the server, using the password of the user;
- a server-side library to perform computations on the encrypted data, namely keyword search;
- an identity provider to verify the public keys of users for authentication if data is shared among multiple users;
- an optional browser extension that allows for detection of client-side code tampering.

A key aspect of Mylar's solution is the ability to do keyword search on the encrypted data by multiple users. To support this, the system uses principals and an access graph to dictate how those principals access the data. Like CryptDB, Mylar uses key chaining: if principal A has access to principal B, Mylar

encrypts the private key of principal B with the public key of principal A. The access graph will have edges going from user principals to object principals and the key chaining method allows for users to access all objects in their access paths.

In the case of keyword search on documents accessed by multiple users, though, there is the following problem. The user wants to search for a keyword $w$ and has his key $uk$, so the client encrypts the keyword with the user's key, $\{w\}_{uk}$, and sends it to the server to perform a search. However, the documents in the server are encrypted with the keys $k_1, ..., k_n$, so, to perform a search, the server needs the client's keyword encrypted with those keys (i.e. it needs $\{w\}_{k_1}, ..., \{w\}_{k_n}$). Mylar introduces the *multi-key search scheme*, a method for *adjusting* a keyword's encryption key to another encryption key, meaning it can transform $\{w\}_{uk}$ into $\{w\}_{k_i}$ without re-encrypting it. This is achieved using cryptographic values called *deltas* (not to be confused with Subversion's *deltas*), where one delta can adjust from one encryption key to another for any keyword. In this scenario, the client computes one delta for each document's key: $\Delta_{uk \to k_i}$, where $i = 1, ..., k$. These deltas are sent along with the keyword encrypted with the user's key, and the server can apply the delta $\Delta_{uk \to k_j}$ to the encrypted keyword $\{w\}_{uk}$ to search in the corresponding document $j$.

### 3.3.3 Discussion

The solutions proposed by Mafdet and Mylar also do not solve our problem of protecting data on a VCS. In these systems the files are not modified, but only retrieved, and that is not the case for VCSes where some of the operations need to modify data on the server. Both Mafdet and Mylar propose interesting ways of encrypting the documents in order to allow search, and support access by multiple users, but they do not solve the problem of needing to modify the encrypted data.

Even though, these systems might serve as inspiration for our solution because we might need techniques to retrieve data from the encrypted repository and we also need support for multiple users with different keys being able to access and collaborate on the same encrypted files in a repository.

## 3.4 Information Deduplication on Ciphered Data

In order to save space in file servers, Information Deduplication techniques started to be used in online storage services such as Dropbox (36) and Google Drive (37). Information Deduplication techniques remove duplicate data for a more efficient storage. Nevertheless, the file servers might be untrusted, so a problem arises if the data to be stored is encrypted and these servers resort to deduplication mechanisms to save space. The following systems ensure that encrypted data can be stored in such file servers without interfering with deduplication.

We start by introducing the concepts of Convergent Encryption and Message-Locked Encryption because they are the basis to support deduplication and also to help understand the DupLESS system explained later in this section.

### 3.4.1 Convergent Encryption and Message-Locked Encryption

Convergent encryption (CE) was proposed in (38) to solve the problem of identifying duplicate encrypted documents. The underlying idea is to define a key for encryption based on the plaintext, so that two equal plaintexts generate the same ciphertext, as long as the encryption method is deterministic. In particular, the key (K) is a hash of the plaintext, such that $K = H(M)$ where $M$ is the plaintext and $H$ is some hash function. The encryption of the plaintext uses that key, such that $C = E_K(M)$ where C is the ciphertext and E is a deterministic encryption function. The ciphering and deciphering are performed on the client and only ciphertexts are present in the server. However, the server only needs to have equivalent ciphertexts for equivalent files so that the deduplication mechanisms can correctly compare and identify duplicated files.

Message-Locked Encryption (MLE) is a cryptographic primitive for encryption schemes that use keys derived from the plaintext messages, like in the case of CE. The work that introduced this primitive (39) studies some of these encryption schemes, which are strongly based on the CE scheme with an improvement to help the server identify the duplicates more efficiently. The improvement is the introduction of tags on the ciphertexts created by the CE scheme. The ciphering and deciphering processes are the same as with CE, but a tag $T$ based on the ciphertext is computed on the server and appended to the ciphertext: $C||T$ where $T = H(C)$. This allows the server to identify duplicates more efficiently because the tag is a summary of the ciphertext and thus the server can compare only the tags to check for equivalence, instead of comparing the entire ciphertexts.

The following are also some MLE schemes based on the CE scheme with tags covered by the study in (39).

**HCE1** is very similar to CE with tags in the sense that it also sets $K = H(M)$ and $C = E_K(M)||T$. The difference is that the tag in this scheme is $T = H(K)$ and so it must be created in the client, instead of the server, because it depends on the key also created in the client only. Therefore, the tag is already appended to ciphertext when it is sent to the server.

The motivation for computing a tag based on the key is to give better performance on the server because it does not have to compute hashes for possibly long ciphertexts. However, this scheme is susceptible to malicious users trying to append arbitrary tags equivalent to tags of different existing ciphertexts. This is known as duplicate faking attacks, and it may possibly induce deduplication mechanisms to delete files that are not duplicate.

**HCE2** is the same as HCE1, but the tag is re-computed in the client when decrypting the ciphertexts received from the server to check if the tag received matches the computed tag. It prevents duplicate faking attacks and the performance overhead introduced by the re-computation is minimal.

**RCE** stands for Randomized CE, and it encrypts the plaintext with a random key $L$ instead of a key based on the plaintext. It adds randomization to the ciphertexts and the server can still detect duplicates because it uses the tags for comparison, instead of the ciphertexts. So the ciphertext will be $C_1 = E_L(M)$. The key $K = H(M)$ is still used for the tag $T = H(K)$ and it will also produce $C_2 = L \oplus K$, such that the ciphertext sent to the server is $C = C_1||C_2||T$. The $C_2$ is necessary for the client to obtain the random key when it decrypts the plaintext.

RCE is the only scheme that can parallelize the encryption of the message and the generation of the key $K$ (to generate the tag). This also means that it is the only scheme that needs to pass over the plaintext only once. RCE also checks the tag upon decrypting the ciphertexts, preventing duplicate faking attacks.

The study concludes that RCE is the fastest scheme while providing the same level of security as HCE2.

### 3.4.2 DupLESS

DupLESS (16) is a system that functions as a wrapper for storage services interfaces to store encrypted documents on those services and still allow the deduplication optimization they implement. It is based on the MLE version of the CE scheme since it is the most prominently used MLE scheme. However, CE is vulnerable to offline brute-force dictionary attacks, so DupLESS brings some security improvements to it.

DupLESS is composed of a key server and a client that acts as a proxy to a storage service interface. The key server is used to authenticate clients, and thus external attackers are not allowed. It also provides keys to cipher the data that are derived from the plaintext in a more complex process than simply hashing the plaintext. The key server acts as single point of control to limit the rate of requests and thereby to prevent brute-force attacks. For the DupLESS client to cipher the data, it uses a variant of MLE, server-aided MLE, which works as the CE scheme but the keys for encryption are obtained from the key server, to ensure authenticity. The client sends the hash of the plaintext to the key server and the key server returns a key derived from the plaintext.

This system also implements some features such as multiple options to limit the rate of requests to the key server to prevent brute-force attacks (e.g. limit the number of requests per user, introduce a delay in the requests), and a function for the user to declare if he wants a file to not be deduplicated, and if so, the DupLESS client generates a randomized (non-deterministic) ciphertext for that document.

The DupLESS client intends to support all operations of the storage service API, so that it can intercept all calls to the interface functions to properly cipher all data before storing it. The encryption methods used to cipher the data are deterministic (AES in CTR mode) so that two equal documents result in the same ciphertext, thus allowing deduplication on the storage service side. The client also ciphers directory and file names to provide more confidentiality, also using deterministic encryption, and encodes the ciphered paths in a character set allowed by the API. To actually store the encrypted data, the client takes the ciphered filename $C_{name}$, creates a ciphertext of the document $C_{data}$ and stores it in a file called "$C_{name}$.data", then encrypts the key used to cipher the document with the user's secret key, producing the ciphertext $C_{key}$, and stores it in a file called "$C_{name}$.key". Any user wanting to retrieve the document has to recompute the ciphered filename to obtain both files, $C_{name}$.data and $C_{name}$.key, and then decipher the $C_{name}$.key with the user's public key to obtain the key needed to decipher the contents of $C_{name}$.data.

Besides the performance overhead of the interactions with the key server, this system also introduces some increase in storage space, although the authors show (16) that this increase is small given

that only a short amount of the documents are required to not be deduplicated. They also show DupLESS fully functioning with Dropbox and Google Drive and their evaluation suggests that the system can be capable of supporting real world workloads.

DupLESS also allows for extensibility of the system to split the documents into blocks and perform deduplication at block level.

### 3.4.3 Discussion

Version Control Systems employ a kind of Information Deduplication by storing only the changes in the files and not the entire files for each revision. Although again, these deduplication systems do not really deal with modifications of the data: if a file is modified on the client, it replaces the previous version on the server. It provides no solution for patching files, as required by the VCS operations.

## Summary

All the systems and techniques we have seen in this chapter provide interesting approaches to the problem of securing data, even though they did not solve the problem of protecting information in a VCS while meeting our goals of minimizing performance and storage space overhead. In section3.1, we saw two approaches that intend to solve our particular problem, although they had severe limitations regarding support for concurrency and multiple users or interference with mechanisms for storage efficiency. We also presented some other systems that dealt with operating on secured data stored in DBMSes, performing searches on encrypted files and allowing information deduplication on encrypted data. They do not apply to our problem, but they still introduced different techniques that positively influenced our solution. The next chapter will explain the architecture and implementation details of our system.

# 4 Trustversion

This chapter presents the design and implementation of our solution for securing data in a Version Control System, called Trustversion. It explains the implementation details of a system providing basic encryption support and a system using append-only encryption. Both represent different approaches for providing data security in a VCS.

## 4.1 Design

The most natural starting point to design Trustversion would be along the lines of Crypto Hook presented in section 3.1.2, which is to encrypt every changed file before committing and decrypt every change during an update. However, as explained before, this approach does not take advantage of Subversion's mechanisms to store *deltas* (differences) of files. Our goal is to maintain all file content encrypted while taking into account the diffing and storage mechanisms of Subversion to introduce minimum storage space overhead in the server.

Our idea follows the same reasoning as Log-Structured File Systems (40) and it is to store all changes on a file like *log-structured changes*, by treating every change as if it was only new data appended to the file. We call this process *append-only encryption* and it is represented in Figure 4.1 for a commit to revision 4. Imagine that Bob created the file `abc.txt` and committed the file to the repository for the first time. This generates revision 1, when the entire file is encrypted by Trustversion and sent to the server. Later, Bob commits additional changes to the `abc.txt` file, creating revision 2. As explained in section 2.2.1, when the client tries to commit a changed file, Subversion generates a *delta* with a description of the changes. In this case, Trustversion produces its own delta, which contains the string "TRUSTV" followed by an encrypted compressed regular diff of the changes introduced in the file. Meaning that, in order to produce the delta for append-only encryption, our system obtains a description of the changes by performing a diff between the file and its previous revision, and then compresses and encrypts the diff. The result is then prepended with a string "TRUSTV" so append-only encryption can identify where a delta in a file begins. This constitutes the Trustversion delta. In Figure 4.1 the delta is depicted as `abc.txt.delta`. With append-only encryption, we append the encrypted delta `abc.txt.delta`, not the `abc.txt` file itself, but to its encrypted version that was committed in the previous revision. The encrypted version of this file was already encrypted in the previous revision using this append-only approach, so we are placing the `abc.txt.delta` delta after a sequence of deltas from previous revisions. Then, Trustversion passes this new constructed file to Subversion as being the `abc.txt` file to perform the commit. The reasoning behind adding the encrypted delta to the encrypted file from last revision and passing it to Subversion, as opposed to passing only the delta, is that Subversion will calculate on its own the file differences independently of Trustversion and generate its own

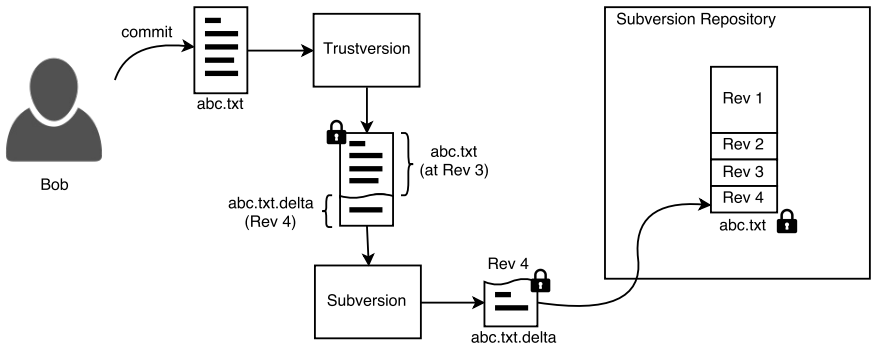


Figure 4.1: Management of file changes during a commit with Trustversion.

delta before sending it to the server. Subversion keeps its own copy of each file from last revision, and uses it to compare with the corresponding file and generate the delta for commit. We want the delta produced by Subversion to indicate that the only changes in the file were the contents of our delta added to the end of the file from the last revision. In other words, we want Subversion's delta to contain only our delta. If we pass just the `abc.txt.delta` Trustversion delta to Subversion, it will believe that to be the `abc.txt` file and that its contents have been completely replaced by this delta. So we send the `abc.txt` file from last revision with the appended delta. Although, the copy from last revision kept by Subversion was already encrypted using append-only encryption. Therefore, we actually pass to Subversion the encrypted version of the file `abc.txt` with the appended Trustversion delta `abc.txt.delta`, otherwise, when Subversion processes the file differences independently, it will generate its own delta including differences besides just our delta. As previously stated, our goal is that Subversion sends only our delta to the server, even though we have no control over its delta generating process. But with this approach, we can make Subversion send only our delta as the only change performed on the file. By doing this for every commit, we trick the server into believing that all the changes are just new binary data (ciphered deltas) appended to binary files (ciphered files). We can see in Figure 4.1 that Bob committing a 4th revision produces a 4th append on the `abc.txt` file from the point of view of the server.

On the other hand, when a client updates an out-of-date file, it will receive deltas from the server indicating only append modifications to the file. We, however, know that these modifications are actually Trustversion ciphered deltas containing the real modifications to the file. Knowing that the Trustversion deltas begin with the string "TRUSTV", to decipher the file using append-only encryption, we identify all the deltas by scanning the file for that string. From the beginning of the file until the first "TRUSTV" is found, the file contains its first version committed, also encrypted, so the first step is to decipher that part and use the resulting plaintext as the base for applying the changes in the subsequent deltas. In the situation presented in Figure 4.1, this would be the portion of the file `abc.txt` corresponding to revision 1. We then sequentially take each delta present in the rest of the file, in the order they appear, and we decipher it, we decompress it and we apply the differences it contains to the `abc.txt` file. We iteratively construct the file since revision 1 until its latest revision. The file is completely decrypted when all the deltas are processed.

On the server the files are stored as pieces of sequential ciphered deltas and, because they are encapsulated in Subversion deltas, the solution does not interfere with the normal operation of the

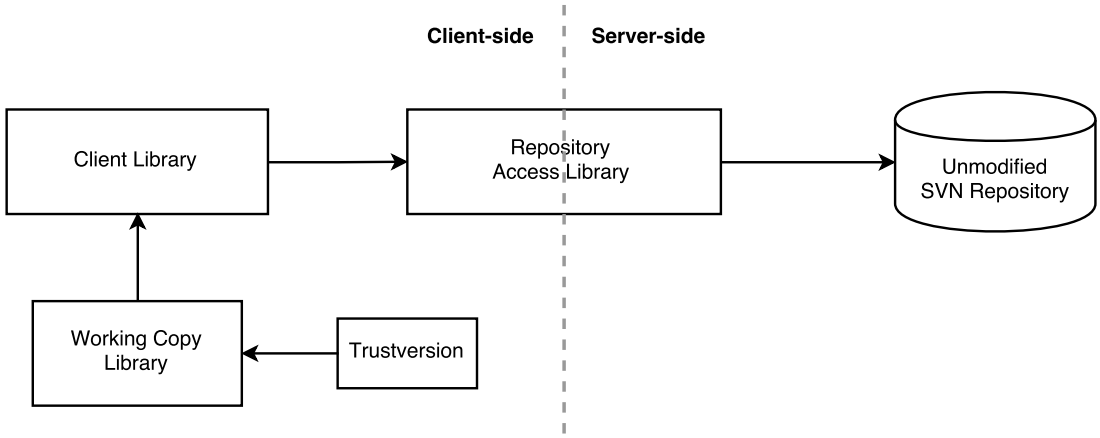


Figure 4.2: Architecture of the Trustversion system.

server. Furthermore, with this solution, the relevant diffing process to obtain the actual file differences is performed by our system in the client on the plaintext files, as opposed to diffing the already simply ciphered files which would cause the generated deltas to be almost as big as the entire ciphered file. And, as we already discussed, trying to diff two different files ciphered with regular encryption methods is rather useless for the purposes of saving storage space.

It may seem like the files on the server will grow infinitely with Trustversion, because even if we removed a big chunk from a file, the change will still be new ciphered information appended to the file. But, aside from the fact that encrypted deltas are slightly bigger than regular deltas, the quantity of information stored in the server will be precisely the same. If we recall the server's algorithm to store changes of files in section 2.2.1, we see that independently of the nature of the changes (even if it is the removal of a big chunk), Subversion always stores it in the server as a delta towards a previous revision describing the change.

Figure 4.2 depicts the architecture of our solution, showing how Trustversion is inserted in the particular case of the Subversion system. It is clear that Trustversion operates only on the client side, working with the Subversion client without interfering with Subversion server's behavior.

Because Trustversion doesn't require the server's implementation to be altered, it benefits from the mechanisms already in place in the server. One important server mechanism is *Representation Sharing*, explained previously in Section 2.2.2. For software project repositories with multiple branches or tags, this mechanism is crucial for saving storage space in the server as it remove as much duplicate information as possible. In order to ensure no impact in this server mechanism, Trustversion uses the same key to cipher all the files within a repository, so that adding full copies of the identical plaintext files produces identical encrypted files. Therefore, Trusversion allows the server to avoid duplicated information while guaranteeing data protection.

## 4.2 Implementation

In this section we will be presenting two separate versions implemented on top of the Subversion Version Control System. The first version provides a basic ciphering functionality to Subversion using the same approach as Crypto Hook, described in section 3.1.2. The second version is the implementation of the append-only encryption system described in Section 4.1.

### 4.2.1 Subversion with Basic Encryption

As discussed before in section 3.1, one possible approach for providing data security in a Subversion VCS is Crypto Hook (34). It is equivalent to a repository user manually ciphering all his files before committing them to the repository.

Since Crypto Hook can only work on a specific Subversion client for Windows, we implemented our own prototype by modifying Subversion in order to see how this approach affects Subversion's mechanisms for saving storage space on the server when applied to large repositories. Its results can also serve as a base for comparison with the results of our solution with append-only encryption.

The prototype was implemented in C, directly modifying the Subversion's source code, specifically the Subversion Client Library (see figure 2.1). We modified only the client-side of Subversion and did not alter the server's implementation. Therefore, it is the Subversion VCS with a transparent built-in file encryption functionality, compatible with any Subversion repository server. The encryption algorithm chosen for the files was the symmetrical encryption AES-256 in CBC mode, with a fixed IV and secret key for the entire repository.

The way it works is very similar to Crypto Hook, although we use some of the Subversion's work to our advantage. The idea is simply to cipher every file in the repository before each commit or update and decipher the files after the commit or update finishes. In our implementation, we cipher only the files that Subversion actually intends to commit because the ciphering process runs only after Subversion determines which files were actually modified. Yet, it is still done before the diffing process is employed by Subversion, guaranteeing the deltas are generated only from ciphered data. These are regular Subversion deltas, produced by diffing two ciphered versions of the same files. These deltas can reach the size of the corresponding files since ciphering two versions of a file with very small differences between them generates completely different encrypted files, rendering the diffing process useless. Since the commit operation does not alter any files, we also keep a copy of each file before ciphering it in order to recover its plaintext version at the end of the commit as opposed to actually deciphering every committed file.

As for the update operation, Subversion cannot determine solely on the client-side which files need to be updated, so we cipher every file in the repository before the update. The need for encrypting the files before the update is due to Subversion receiving deltas that were generated against the ciphered versions of the files and applying these deltas directly on them. Since Subversion expects all files in the working copy to be ciphered, the update operation will not function properly if the files are all in plaintext, so they must already be ciphered when the deltas are applied. Because the update operation typically

alters files, we also have to decipher every file after the update, as opposed to saving and restoring the plaintext version as we do with the commit operation.

This implementation does not alter in any way the Subversion's diffing mechanisms, it just simply encrypts the entire files every time an operation is performed on them. As we mentioned before, in this case, even a minor modification between two plaintext versions of a file will produce completely different encrypted versions of that same file. And being encrypted data, the deltas generated for this file can be as big as the encrypted file itself.

### 4.2.2 Append-only Encryption

With the implementation of our solution we took a different approach compared to the prototype described in Section 4.2.1. We did not modify the Subversion code at all, and, instead, we implemented wrappers for the SVN operations. Although it removes some transparency for the user, that would have to use the Trustversion system explicitly, it can work independently of the version of Subversion the user has installed. This means that Trustversion is compatible, not only with any Subversion server, but also with any Subversion client.

These wrappers are a set of Perl programs, each program performing one SVN operation, implementing the append-only encryption approach. The encryption algorithm used in this approach is the symmetrical encryption AES-256 in CBC mode. To simplify the automation of the tests to our solution, we defined a fixed IV and secret key for the entire repository, although, with a small change these wrappers could accept a secret key from user input. They fully support the most used SVN operations – commit, update, add, remove and status – and they work as follows. They perform append-only encryption operations on the repository files before and after they call the corresponding Subversion operation, delegating the actual behavior of the operation to Subversion.

As we have described in the section 4.1, Trustversion generates its own deltas to append files before passing them to Subversion. Due to running independently of Subversion, in order for our system to be able to generate its own deltas, we have to keep track of files' modifications, so, for that purpose, we maintain a copy of the plaintext and the ciphered versions of every file in the repository. They are stored inside a directory named `.trustversion`, where the plaintext copies are stored inside the directory `.trustversion/plaintext` and the ciphered copies are stored inside the directory `.trustversion/ciphered`. A plaintext copy of a repository file in our context refers to the unciphered version of that file from the last time it was committed. Meaning that, on each commit, we make a new copy of the file while unciphered, replacing the version from last revision. The same applies to the ciphered copy of a repository file, which is the ciphered version of the file, encrypted using append-only encryption, from the last time it was committed. So, on every commit, after performing append-only encryption on the file, we also make a new copy of this ciphered file and replace the previous ciphered copy. Therefore, for every file in the repository, there is a plaintext copy in the `.trustversion/plaintext` directory and a ciphered copy in the `.trustversion/ciphered` directory, in which both copies correspond to the exact same revision. Since the plaintext copies contain the files' contents from the last revision they were committed, we use them to identify which files have been modified by the user since the last commit by comparing each one to its plaintext copy. We also use them to help build our append-only

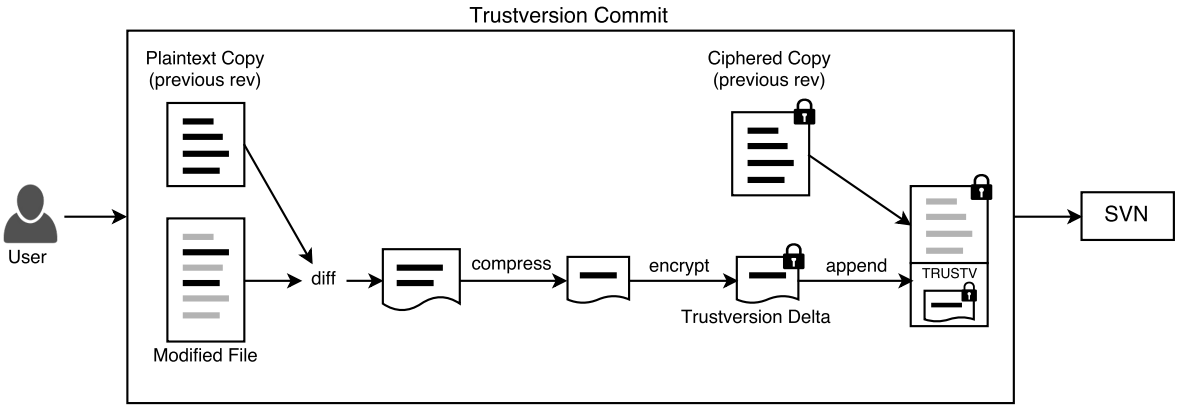


Figure 4.3: Implementation of the management of a modified file during a Trustversion commit.

deltas by diffing the modified file against its plaintext copy. The ciphered copies helps us build new ciphered files on each commit, using append-only encryption, and to identify modified ciphered files after each update, in the same way we identify modified plaintext files. We also use the ciphered copies to help rebuild the corresponding plaintext versions after being modified in the update operation.

When performing a commit using Trustversion, we start by traversing the repository's directory structure and comparing every file to its plaintext copy to identify which files were modified. For every unmodified file we simply replace it by its ciphered copy, so that it also appears unmodified to Subversion when we invoke Subversion's commit. When a modified file is encountered, for example `abc.txt`, we create a delta by invoking the operating system command *diff* (41) between the modified file and its plaintext copy, generating a file `abc.txt.diff` describing the changes in the *diff format* (42). We then take the diff file and compress it using most typical Zip algorithm *Deflate*, generating the zip file `abc.txt.diff.zip`. Finally, we cipher this zip file using the AES-256-CBC symmetrical encryption, we insert the string `TRUSTV` at the beginning and store the result in the delta file `abc.txt.diff.enc`. As we have seen in section 4.1, the string `TRUSTV` helps us later to identify the delta in the ciphered file. This describes the process of creating a Trustversion delta for append-only encryption and can be seen in Figure 4.3. After obtaining the delta file `abc.txt.diff.enc`, we make a new copy of the plaintext file that we store in the `.trustversion/plaintext` directory, we append the delta file to the ciphered copy of the file stored in the `.trustversion/ciphered` directory and, lastly, we replace the current plaintext file by the ciphered copy already including the appended ciphered delta. Therefore, depending on whether the repository files have been modified since last revision or not, all of the repository files are, respectively, either encrypted with append-only encryption, or replaced by their ciphered copies that were encrypted with append-only encryption in previous revisions. After the entire repository is traversed and the all the files are encrypted, the Subversion's commit is invoked to perform the actual commit with a repository containing only ciphered files. When Subversion's commit ends, we simply restore the plaintext copies by replacing all the currently ciphered files by its plaintext copies.

Intuitively, performing an update operation using Trustversion is the opposite process. We also traverse the repository's directory structure, although in this case we start by replacing every file by its append-only encryption ciphered copy, and then invoke Subversion's update on the ciphered repository.

Only after Subversion's update finishes we traverse the repository with the purpose of finding which files were modified by the update. To achieve this we compare the currently ciphered files with its corresponding ciphered copies. Unmodified files are simply replaced by its plaintext copy, but modified files must be scanned for Trustversion deltas and processed in order to rebuild the plaintext files. As we have described in Section 4.1, a file that has been encrypted with append-only encryption is its ciphered version on its first revision followed by a sequence of Trustversion deltas describing all the changes until now. This way, the corresponding plaintext file can be totally constructed from zero. But since we keep a copy of the ciphered and plaintext versions of each file on its last commit, we can use them to process only part of the encrypted file. We start by identifying where the new data in the encrypted file begins. We take the ciphered copy from the `.trustversion/ciphered` directory, since it contains the last ciphered version we have of the file, and use the file byte size of the copy as an offset for the updated ciphered file. Being an append-only approach, we know that all the new data must by exclusively at the end of file, so the part of the updated file from the beginning until that offset contains exactly the same content as the ciphered copy, and the new data added by the update operation must start at the offset. Knowing that a file was modified by Subversion's update, we also know that it must have at least one new delta appended, but if the file was out-of-date by multiples revisions, the update may have added multiple deltas to the end of the file. Therefore, when we process the updated file, we start at the offset, scanning for the string `TRUSTV` to capture all the deltas added in the update. Every time we find the string `TRUSTV`, we retrieve data from the file until the next `TRUSTV` appears or until the end of the file. These pieces of data are the Trustversion deltas. For each delta we retrieve from the updated file, we store it in the `abc.txt.diff.enc` file and we decipher the encrypted delta file resulting in a compressed diff that we store in the `abc.txt.diff.zip` file. We then decompress it and invoke the operating system command *patch*, which patches data in the *diff unified format* to plaintext files, to apply the changes in the diff file `abc.txt.diff` to the plaintext copy of the file `abc.txt`. As we have said above, the plaintext and ciphered copies of a file correspond to the same revision. If we used the ciphered copy to process only the deltas added after the last revision, we can safely apply these changes to the plaintext copy in the `.trustversion/plaintext` directory, since it is from the same revision. Using this process, we can sequentially apply all the new Trustversion deltas in the file and, thus, avoid a complete rebuild of every file updated by Subversion.

A file that has only been added to the repository does not contain deltas. When a file is added using the append-only system add, we copy its plaintext version to the `.trustversion/plaintext` directory, we compress and encrypt the file and copy this ciphered version to the `.trustversion/ciphered` directory. We then invoke Subversion's add operation on the encrypted file and after the operation finishes, we recover its plaintext version. This add operation provides the base file for all subsequent appended deltas.

The append-only system remove operation is the most simple operation since we only need to delete the plaintext and ciphered copies of the files to be removed before we delegate the rest of the work to Subversion's remove operation. If at anytime a removed file is recovered from the Subversion server, we can completely rebuild the plaintext version from the ciphered file, thus recovering both the plaintext and ciphered copies.

The status operation also relies on the `svn status` command, using its output to know which files have been added or removed, and print that exact information in the same output format as the

`svn status`. The remaining files in the repository are all compared with its plaintext copies stored in the `.trustversion/plaintext` directory, and only the files that have differences towards its plaintext copies are presented as modified. The regular Subversion's status operation will show all files in the repository as modified, even if they're not, because Subversion knows only their ciphered versions. When performing the Subversion status operation on a totally unencrypted repository, Subversion compares every file to its own copy from the last commited revision, which is the ciphered version of the file, thus finding that every file in the repository is different from its own copy, because it is in plaintext, and declares the file as modified.

## Summary

This constitutes the implementation of our solution, Trustversion. Its design, also detailed in this chapter, introduced that notion of an append-only encryption solution that cooperates with the mechanisms for storage space efficiency in an unmodified server. As mentioned above, we implemented this solution with the Subversion VCS using SVN wrappers, but also altered the Subversion system to implement an approach of basic encryption on the repository files. We have put these systems to test with large repositories in order to evaluate the results and discuss their impact on big sets of data. The evaluation and discussion of the obtained results are in the next chapter.

# 5 Evaluation

In this chapter we introduce the experimental evaluation to both approaches described in the previous chapter: Subversion with basic encryption and Subversion with append-only encryption. We present and compare its results to analyze the advantages of our solution. We also present the work involved in acquiring test cases and achieving these results.

## 5.1 Acquiring and Recreating Repositories

To really capture the impact Trustversion can have on a repository in terms of time and storage space overhead, we applied our system to repositories with a great number of files and/or revisions. We were also interested in practical and typical cases for repositories so we decided to use publicly accessible repositories of big software projects.

For us to apply both implementations on an already existing repository, we had to be able to recreate each revision since the beginning of the repository. In order to do this, we first implemented a repository "crawler" that retrieves the entire revision history of public repositories. Then we implemented a repository "replayer" that can reproduce the entire history of a repository revision by revision. We describe both the crawler and the replayer in the following sections.

### 5.1.1 Repository Crawler

As mentioned above, the repository crawler is a script that tries to capture all the changes made in a repository since revision 0. This is only possible with publicly available repositories with unlimited read access, so we can have the script running autonomously, continuously invoking SVN commands on the repository, without constantly requiring authentication or limit in any way the information we want to retrieve. The repository crawler script was developed in Perl, consisting of 660 lines of code, and runs with any typical Perl installation, without requiring any extra Perl modules. It also relies on having an unmodified Subversion installed.

The first step of this script is to invoke the `svn log` command on the repository, which gives us the list of all the revisions and its authors, dates and commit messages. We want to gather as much information as possible to be able to later use the replayer to recreate a new repository equivalent to the original repository. We then iteratively retrieve the repository's history. For each revision, we use the `svn update` and `svn diff` commands to download all files in the repository on that revision and obtain a list of which files were modified, added and removed. We store this along with the revision's

number, author, date and commit messages that we gathered previously. This way we keep the complete repository state for every revision.

One possible way of storing all this information was to create a directory tree representing the repository, where each revision would be a directory that would keep all the revision's files and data. Although, this would be a very inefficient way of storing the repository's history given that we have all of the repository's files for each revision, so most of them would be identical copies among multiple directories. Therefore, we decided a more efficient way to save this information would be in another Subversion repository. So the script creates an empty local repository and each commit to the repository contains a revision from the original repository, meaning the revision's files and the revision's metadata.

The importance of retrieving a repository's history and saving it locally is that it allows us to *replay* the original repository as many times as we want and in controlled environments where the results are not affected nor dependent of the accessibility or availability of a public remote server. In fact, while running the crawler with public repositories, we experienced problems regarding availability and connectivity to the servers. Some of the problems we faced included:

- Servers limiting the number of VCS operations per minute;
- Dropped connections during VCS operations involving large data transfers;
- Network connectivity problems;
- Network latency.

We adapted the script to circumvent some of the issues. For example, we added a delay between consecutive SVN command calls to avoid reaching the limits imposed by repository servers. We also modified the script to restart from the point where the connections were dropped. Dealing with these issues during the execution of the crawler, whose goal is to locally store the repository's history, allows us to later recreate repositories using our solution with more reliable results and more accurate measurements. The phase for reproducing repositories is performed by the "replayer" script described in the next section.

### 5.1.2 Repository Replayer

The repository replayer script is also developed in Perl, consisting of 420 lines of code, and requiring no extra Perl module to run. As mentioned above, the repository replayer is a script capable of recreating an entire repository based on the information stored by the repository crawler. This script aims to generate a repository identical to its publicly accessible version at the point in time that the crawler was able to capture. We not only reproduce the file changes from each revision to the new repository, we also copy the authors and the commit messages. The goal of this script is to automatically interact with a repository using different implementations of Subversion while simulating the typical usage of a VCS with different users in real projects. We also want to gather data in order to analyze the impact of our solution in comparison to other implementations of Subversion and obtain results mainly concerning the execution time of each VCS operation and the disk space occupied in each revision.

| Project Name | Total Revisions | Files Size (MB) | Repository Size (MB) | Total Users |
|---|---|---|---|---|
| Ant | 12480 | 923,32 | 298,47 | 45 |
| CloudStack | 415 | 4265,64 | 592,0 | 21 |
| CouchDB | 4081 | 106,81 | 63,69 | 23 |
| Perl | 11628 | 660,65 | 368,48 | 45 |

Table 5.1: Storage space usage on the Ant repository.

To replay the repository, we apply and commit the file changes revision by revision starting from the first revision, similarly to the crawler. As explained in the previous section, the chosen method for the crawler to store the information was a Subversion repository. This means that the replayer always requires a regular unmodified SVN installed to recover the information stored in the crawler's repository. So the replayer uses two versions of the Subversion system, the regular SVN to get the information saved by the crawler and a second version depending on the system we choose to replay the repository with. If we want to replay a repository using the append-only encryption system, the replayer uses both the regular SVN and the append-only encryption system, where the latter is the system used to perform the VCS operations on the new repository that simulate the changes made in the original repository.

To replicate a revision, we start by using the regular SVN to perform an `svn update` and get the files from the crawler's repository on that particular revision. Besides the repository's files, we also get a list of which files were modified that contains the information about which of those files (or directories) were added or removed in this revision. Using the chosen SVN system, for example append-only encryption, we perform an add or remove operation on the corresponding files or directories. We also use the information about the author and commit message of the original revision and perform a Trustversion commit of the added, removed and modified files using the original commit message and pretending to be the original author. This way we get an identical revision on the new repository. We apply this process orderly for every revision of the original repository to replicate its entire history. Because both Subversion with basic encryption and Subversion with append-only encryption were implemented using fixed secret keys, this script could run autonomously, without requiring a password as input, although it could easily be adapted to provide a password for every VCS operation.

During the execution of this script, we also record some data regarding the time of execution of every VCS operation and the occupied space by the repository both in the client and the server, so we could process this data and analyze the impact of each system we implemented. The results are presented and discussed in the following section.

### 5.1.3 Collected Datasets

In this section we present the repositories we collected from public Subversion servers using the repository crawler script. All of the repositories are know software projects, and we display them along with some of its characteristics in the following table. The projects' repositories information shown in this table includes the number of revisions in the repository, the total size of the files in the repository, the total size of the repository in the server and the number of users that committed changes in the project.

| Project Name | Ant | CloudStack | CouchDB | Perl |
|---|---|---|---|---|
| Total Revisions | 12480 | 415 | 4081 | 11628 |
| Files Total Size (MB) | 923,32 | 4265,64 | 106,81 | 660,65 |
| Number of Files | 69560 | 126993 | 11427 | 70300 |
| Repository Size with Subversion (MB) | 298,47 | 592,0 | 63,69 | 368,48 |
| Repository Size with Basic Encryption (MB) | 3524,24 | 1395,0 | 188,19 | 2313,89 |
| Repository Size with Append-only Encryption (MB) | 680,96 | 1198,06 | 117,1 | 603,54 |
| % Storage Overhead with Basic Encryption | 1080,77% | 135,64% | 195,48% | 527,96% |
| % Storage Overhead with Append-only Encryption | 128,15% | 102,38% | 83,86% | 63,79% |
| Average File Size (B) | 12,79 | 34,27 | 9,12 | 8,95 |
| Average Commit Time with Subversion (s) | 1,24 | 1,71 | 0,26 | 0,61 |
| Average Commit Time with Basic Encryption (s) | 61,34 | 23,04 | 0,3 | 1,64 |
| Average Commit Time with Append-only Encryption (s) | 171,15 | 2760,06 | 4,13 | 76.,68 |
| Total Add Operations | 1816 | 87 | 451 | 1526 |
| Total Remove Operations | 545 | 19 | 177 | 283 |
| % Add Operations | 14,55% | 20,96% | 11,05% | 13,12% |
| % Remove Operations | 4,37% | 4,58% | 4,34% | 2,434% |

Table 5.2: Summary of the results on the 4 software projects.

## 5.2 System Performance

We evaluated 4 big software projects with 3 different systems: the unmodified Subversion, the modified Subversion with basic encryption and Trustversion, our solution. Our goal was to obtain data using the regular Subversion system to serve as a baseline for comparison and obtain data using the other two systems to observe the overhead each system introduces by adding encryption on this VCS. It was also our goal to empirically show how our solution is preferable to simply ciphering all the files, the approach provided by the Subversion with basic encryption, described in section 4.2.1.

### 5.2.1 Evaluation Environment

All the tests with the replayer script using the three systems were performed in VirtualBox virtual machines running the Debian Linux distribution Each virtual machine was configured with one single-core CPU and 2GB of RAM. Every test required 2 virtual machines, one serving as the SVN client and the other serving as the SVN server, and every set of client-server VMs ran exclusively one test at a time. A test comprises one complete repository replay using one specific system. The results of each test are presented in the following section 5.2.3.

The software version of the unmodified Subversion system used, which is that same system our solution relies upon, is Subversion 1.8. The Subversion system with basic encryption is also based on the Subversion 1.8.

### 5.2.2 Evaluated Datasets

We evaluated the repositories collected by the repository crawler script. These are 4 known software projects, namely Ant (22), CloudStack (19), CouchDB (20) and Perl (21), whose characteristics we layed

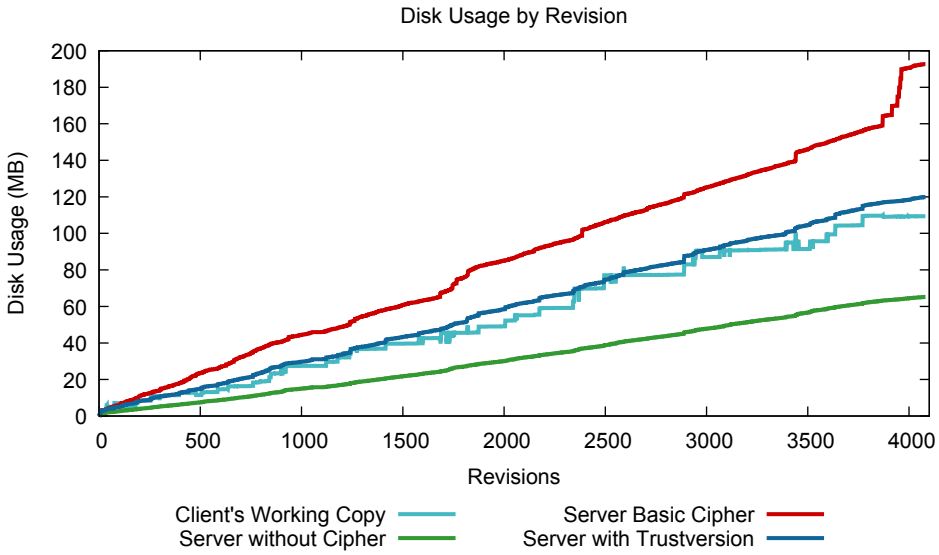


Figure 5.1: Storage space usage on the CouchDB repository.

out in the 5.1.3. We replayed each repository 3 times: using the unaltered Subversion system, using the Subversion with basic encryption (section 4.2.1) and using the append-only encryption system (section 4.2.2). During every repository replay, we collected data regarding the execution times of each VCS operation performed and the storage space usage on each revision. A summary of the data we collected for the 4 software projects is presented in Table 5.2.

This gives a summarized overview of the measurements we used while evaluating each project. The next section discusses in detail these results.

### 5.2.3 Results

This section discusses the results of the tests we performed on 4 repositories containing big software projects. These software projects and a summary of their results are displayed in the table above. The software projects are the following: CloudStack (19) with 415 revisions, CouchDB (20) with 4081 revisions, Perl (21) with 11628 revisions and Ant (22) with 12480 revisions.

The CouchDB project shows good results regarding Trustversion's storage space usage, as we can see in Figure 5.1. The Figure 5.7 shows the space occupied on each revision by CouchDB in the client's working copy, which constitutes the total size of all project files, and the space occupied in the server when replaying the repository with no encryption (regular Subversion), basic encryption and append-only encryption. In this graph we can see clearly how much more storage space an append-only encryption can save in comparison to a basic encryption. CouchDB's repository has a total of 4081 revisions with 11427 files on the last revision, with an average of 9 bytes per file. And at the end of the repository's history, the occupied size on the server with a replay using append-only encryption is approx. 117MB while the occupied size with the basic encryption system is approx. 188MB. This means that, by using our solution, we saved 71MB in storage space in the server, which is more than the total size of the CouchDB repository without any encryption. Our solution introduced 83,86% of storage

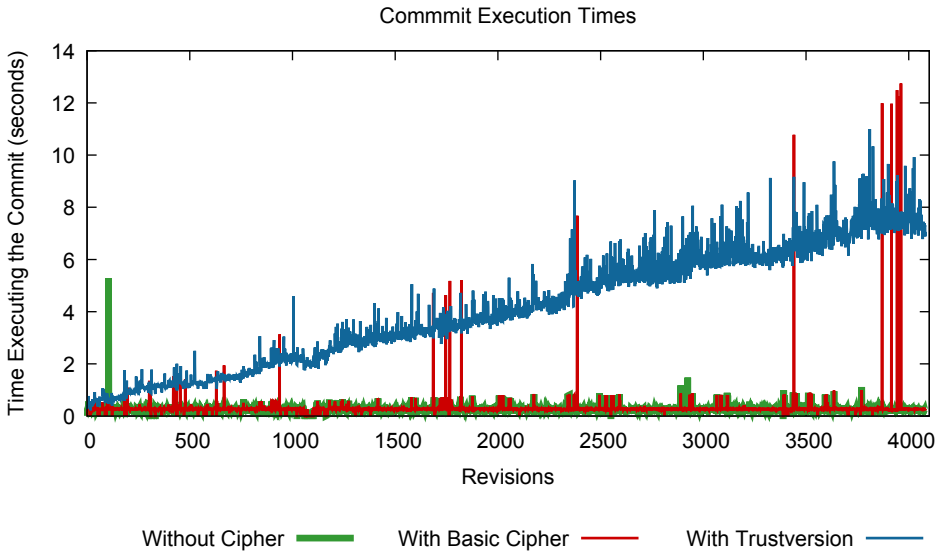


Figure 5.2: Execution time of every commit operation in the CouchDB repository.

space overhead in relation the regular Subversion system, while the basic encryption system introduced 195,48% overhead, constituting almost 3 times the size of the repository without encryption.

In the commit execution times graph in Figure 5.2, we see results that are not so great for our solution. Looking at the graph, we can immediately see that the append-only encryption has the worst performance results of all three systems. Although, we can also see that these results for Trustversion are proportional to the results in the storage space graph in Figure 5.1. In fact, they are proportional to the disk usage data on the client, because Trustversion is affected by the increase in the files size in the client, but also greatly affected by the increase in the number of files. Since Trustversion runs separately from Subversion, it does not benefit from Subversion's mechanisms to quickly identify modified files. While Trustversion identifies modified files by comparing every file in the repository to its plaintext copy kept by Trustversion, Subversion stores additional information about the files, namely the modification time and the file size, to identify which files were not modified. This way, Subversion only actually compares the files that have a different modification time or file size from the information that it has stored. Trustversion's approach is not as time efficient but is more reliable, since tampering with a file's modification time or file size wouldn't affect Trustversion's behavior. Besides, even though our solution has an average of commit execution time approx. 17 times worse than using no encryption and and 12 times worse than using basic encryption, the maximum high of 13 seconds to perform a commit belongs to the basic encryption system. There are some revisions throughout the repository's history where basic encryption has had worse performance than append-only encryption. We can also see that these situations where basic encryption provides such high results for commit execution times are also proportional to the file size in the repository. These isolated situations mainly represent revisions where directory trees are removed through the remove operation. If that directory has had files modified in the past, this particular change requires subversion to perform an `svn update` on the entire repository before committing the removal of the directory. Since the replay script is the only committer in the repository, the working copy is always up to date and the update operation does not actually have any files to update, but Subversion still updates the repository's metadata with information from the server.

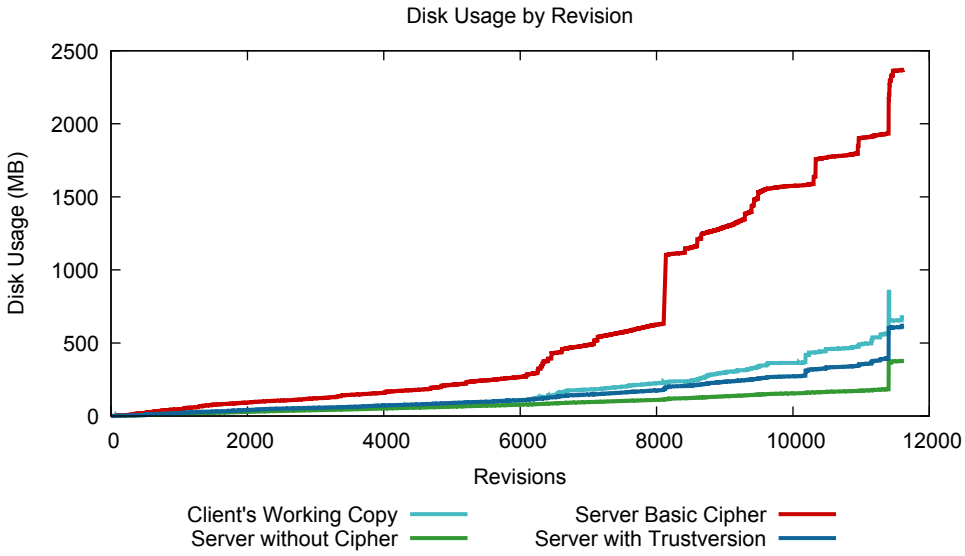


Figure 5.3: Storage space usage on the Perl repository.

After this update of metadata also performed using basic encryption, Subversion believes that all the files in the repository are modified, so it processes every file when the commit for the directory removal is performed, even though it does not actually commit any files because they weren't really modified. But processing every file in the repository looking for changes is what causes a high execution time overhead for theses particular situations.

Overall, waiting an average of 4,13 seconds for a Trustversion commit for 71MB in saved storage space in the server is a reasonable tradeoff.

The results for the Perl project are a good example of what happens with each system as the number of revisions increases. The Perl project has 11628 revisions, 3 times more revisions than CouchDB, and seeing the storage space occupied on the server with each system in Figure 5.3, it is clear how over time a system without the append-only encryption approach for storage efficiency affects the server. With the same number of revisions as CouchDB (4081 revisions), it is already visible in Figure 5.3 that Perl replayed with Trustversion is more efficient than using simple encryption. In fact, at that point in the repository's history, Trustversion has only 40,5% storage space overhead while Subversion with basic encryption has 219,5% storage space overhead, translating to 3 times more occupied space in the server than the unencrypted repository. But the significant evidence of Trustversion's ability to keep files encrypted and still be efficient with storage usage starts around revision 6000. Between revisions 6091 and 7136, most revisions were mainly constituted of modifications on existing files, and some of the commits performed included an average of 1000 modified files per commit. In that interval, there is an approx. 260MB increase in occupied space with simple encryption, whereas with Trustversion there is only an approx. 40MB increase. This is corroborated by the disk usage graph that shows us a clear impact on the server's disk storage when using Subversion with basic encryption and no visibly impact when using Trustversion. An even more accentuated impact of the same situation is visible between revisions 8105 and 8138, where each one of those 33 revisions is a commit of modifications to approx. 1200 existing files. With just 33 consecutive commits of modifications only, the server's occupied space

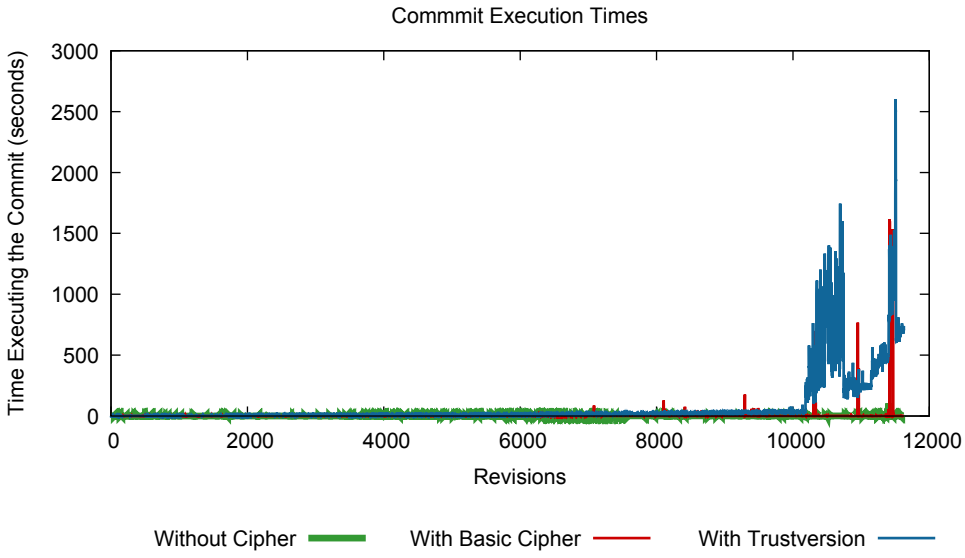


Figure 5.4: Execution time of every commit operation in the Perl repository.

jumped 470,08MB using Subversion with basic encryption while the Trustversion system caused an increase of only 18,78MB in the server. This is the type of situation where using Trustversion for data security is the most beneficial. When a revision or set of revisions are mostly constituted of modifications on existing files rather then added files, Trustversion is much more efficient than a simple approach for encrypting the files because, as we explained in detail in section 4.1, the resulting revision *delta* of a modified file using Trustversion contains only the actual changes to the plaintext file, as opposed to a system with basic encryption whose resulting *delta* is the entire encrypted file. Multiplying this effect by 1200 files on 33 consecutive revisions creates the huge impact on the server that is so clearly visible in figure 5.3. By the end of the repository, Trustversion is able to achieve an overhead in storage space of 63,79% in relation to an unencrypted repository while Subversion with basic encryption reaches an overhead of 527,96%, corresponding to approx. 3 times more space occupied in the server than when using Trustversion.

Unfortunately, Trustversion's performance regarding the execution time of the commit operations suffers with the increase of files and repository size. At the end of the repository, the Perl project contains 70300 files with an average size of 8,95 bytes per file. It contains almost 7 times more files than CouchDB. This is mostly why in the last revisions of the repository, Perl's execution times, shown in figure 5.4, are so much worse than CouchDB's execution times (figure 5.2). Although, until revision 10185, Trustversion performs commits with an average execution time of 12,18 seconds, just slightly higher than CouchDB's worst execution times. If we recall the discussion about Perl's disk usage graph, most of the revisions are primarily comprised of modifications on existing files. Therefore, the number of files in the repository did not change much during these revisions, and neither did the repository's size as we can see in that same graph, figure 5.3, so the execution times were not affected by these changes. But towards the end of Perl's repository, starting from revision 10185, a great number of files were added and the great majority of the add operations were *tags* and *branches*. As we have also explained before in section 2.2.2, adding tags and branches can introduce a huge increase in number of files in the repository with minimal impact in the server's storage space. This is the reason why we only

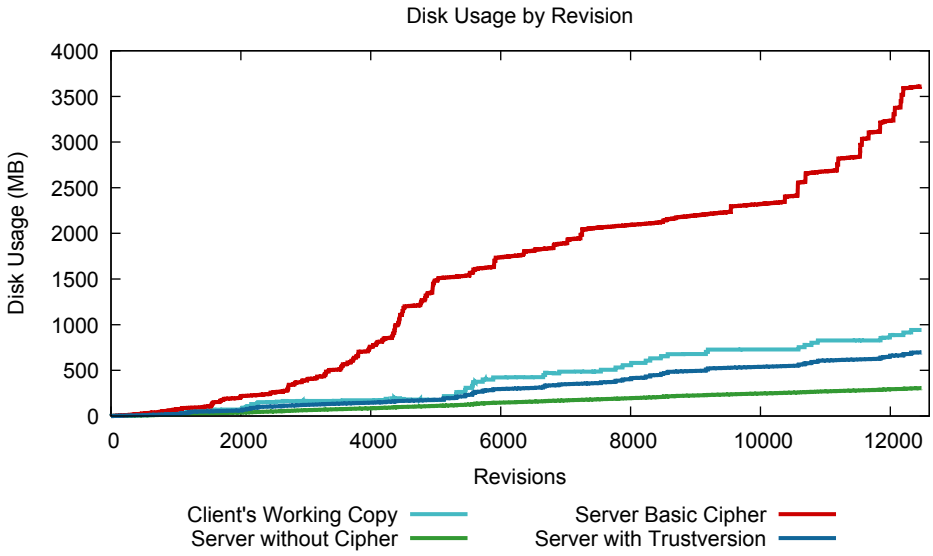


Figure 5.5: Storage space usage on the Ant repository.

see a small storage usage increase in the disk usage graph in the revisions following revision 10185, even though those revision added up to 7736 files to the repository. The increase in repository files is what caused the spikes we see in figure 5.4. The decrease in execution times we also see around revision 11000 is due to some tags being removed from the repository, decreasing the number of files to be processed by Trustversion.

At last the Ant project also provided us with very good results regarding Trustversion's storage space efficiency, shown in figure 5.5. Ant is a repository characterized by having a high rate of modifications in existing files, where 81,08% of all revisions contained only file modifications and just 14,55% of all revisions included add operations. We have 12480 revisions in Ant's repository, just 852 more revisions than with Perl's repository and 3 times the revisions of CouchDB. As we've already seen with Perl, Ant shows us how Trustversion can be really beneficial for saving disk space in the server in the long-term with typical software projects, where most revisions include only small modifications to existing files. With just 3 times more revisions than CouchDB, we have a clear vision of the huge impact a simple encryption on the repository can have over time. But Ant is a good case scenario for Trustversion, given that Ant reached the same number of revisions as CouchDB (4081) with 71,35% storage space overhead using Trustversion but 817,96% storage space overhead with Subversion with basic encryption. This means that, while the repository occupied less than twice the space in the server with Trustversion than with Subversion without encryption, Subversion with basic encryption introduced 8 times more occupied space, just until revision 4081. Although, we can immediately see in figure 5.5 that at the end of the repository the difference in disk usage between systems is far greater. On the last revision, the unencrypted repository in the server occupies 298MB, the Trustversion repository occupies 680MB and the repository with basic encryption occupies 3,4GB. Therefore, in comparison with using no encryption, Trustversion achieves an overhead of 128,15% that is still significantly better than the overhead of 1080,77% introduced by Subversion with basic encryption. For approximately the same number of revisions as Perl, the Ant repository did not provide results as good as Perl's results, having significantly more overhead. Ant had approx. the same number of files as Perl at the end of the repository, but 4

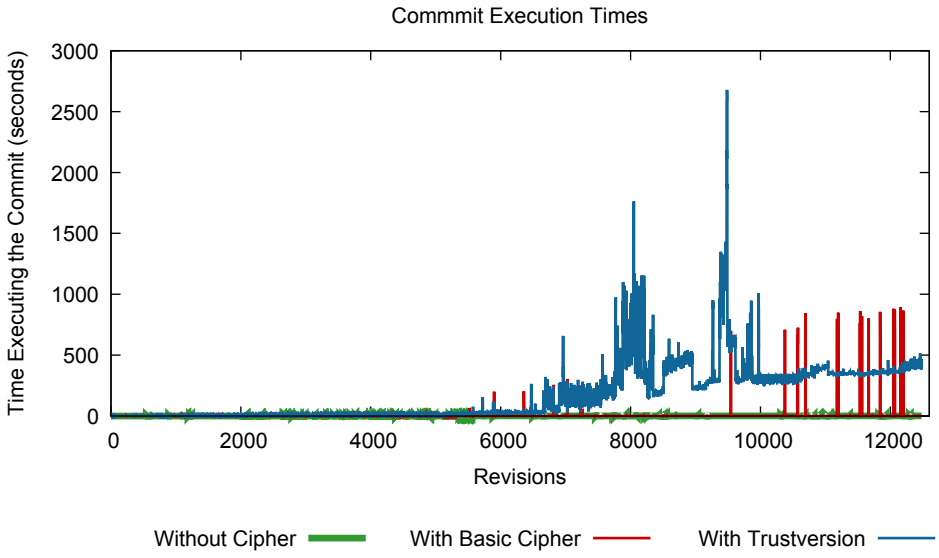


Figure 5.6: Execution time of every commit operation in the Ant repository.

bytes more per file and 1,4% more revisions with add operations than Perl, meaning that in average an add operation for the same number of files would be bigger in file size in Ant. Furthermore, as we can see in figure 5.5 the early and huge impact this repository had on Subversion with basic encryption means that Ant had a lot more file changes throughout the repository's history than Perl (figure 5.3), since modifications on existing files is what marks the biggest differences in storage space overhead between systems. Therefore, Ant introduces a bigger overhead than Perl, but it also has more file changes and a bigger average file size, so the additional overhead is expected.

On the other hand, the performance in terms of execution time for commit operations in this repository is not very good. The figure 5.6 shows each commit operation execution time using the three systems. We can see significant increases in the execution times at approx. revision 6700, then again around revision 7800 and then again around revision 9400. All these cases are reflections of increases in the number of files and size of the repository and large changes on the already existing files. We should also not that, since Trustversion runs as a wrapper for Subversion, the execution time of a Trustversion commit is the composition of Trustversion's time performance and Subversion's time performance. This is why when big changes are made to the repository, or a big number of files are added, the execution time is greatly magnified. This is also what causes the spikes in the time overhead, because when large changes are made to the repository, both Trustversion's commit and Subversion's commit take longer to run, but with small changes, it is mostly Trustversion introducing the performance overhead. Additionally, we can see towards the end of the repository replay that with Subversion with basic encryption we have some revisions with high execution time overhead. In this case, it involves revisions that delete entire directory structures and, as we've discussed earlier, this causes significantly high commit execution times for the basic encryption system. Nevertheless, Subversion with basic encryption has consistently better performance than Trustversion. Throughout the repository replay, Subversion with basic encryption had an average execution time of 3,32 seconds per commit while Trustversion reached an average of 170,40 seconds per commit, significantly worse than the performance on an unencrypted repository with an average of 0,67 seconds per commit.

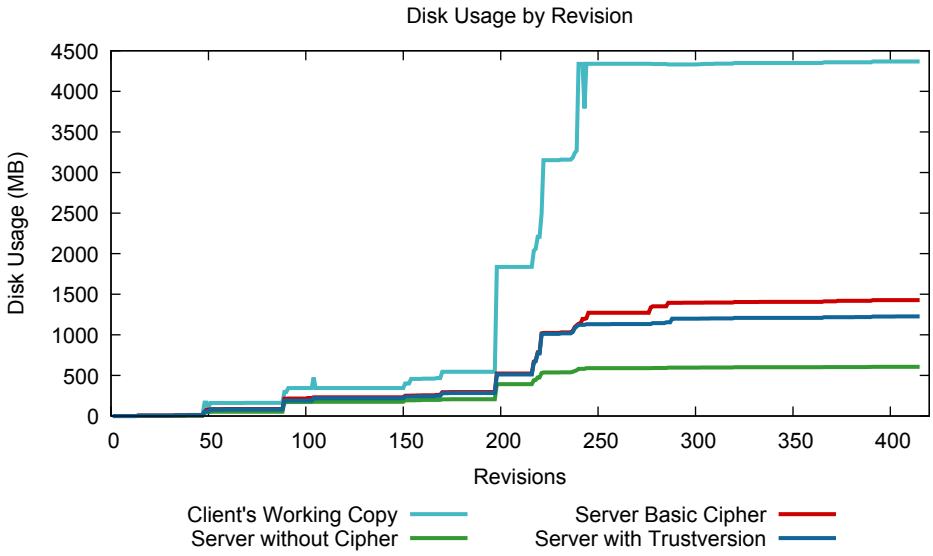


Figure 5.7: Storage space usage on the CloudStack repository.

The CloudStack project is an interesting repository for our evaluation because, even though it has very few revisions in comparison with the other projects we tested, it is the biggest repository in terms of space and in number of files. The Figure 5.7 shows the space occupied on each revision by CloudStack where we can see that, since revision 240, the repository keeps almost 4,5GB of files. On the last revision, the approximately 4,5GB of information are distributed among approx. 127000 files with an average of 34 bytes per file. Meaning that this repository is larger than all the other repositories we've seen both in the number of files and in the average file size. In reality, CloudStack is the worst case scenario for Trustversion, not only due to the big number of files and file size, but also because approx. 20% of the changes committed were added files and the true advantages from using Trustversion instead of a simple encryption arise from multiple modifications to the existing files. Adding files using the append-only encryption system is simply sending totally encrypted files, and the same happens with Subversion with basic encryption. So in a repository with a higher rate of add operations and lower rate of file modifications, the gain of using Trustversion will not be as significant, and we can see that in Figure 5.7. While Trustversion still performs better than Subversion with basic encryption in terms of saving storage space, the difference is very small. With append-only encryption, the occupied space in the server is only 197MB less than using basic encryption, constituting only 33% of the space occupied by the repository with no encryption. Furthermore, while our solution introduced 102% of storage space overhead in comparison to using Subversion without encryption, the second worst result for this measurement, the basic encryption Subversion introduced only 136% of storage space overhead, its best result for this measurement. This further shows how Trustversion is not ideal for repositories like this.

Additionally, because this is also a software project with a great number of branches, most of the space usage efficiency we see in Figure 5.7 on both encryption system is due the *Representation Sharing* mechanism we described in section 2.2.2, employed by the Subversion's server that we have not altered. In this repository, the number of duplicated information is such that the server is able to save approx. 3GB in the encrypted files and 4GB in the unencrypted files.

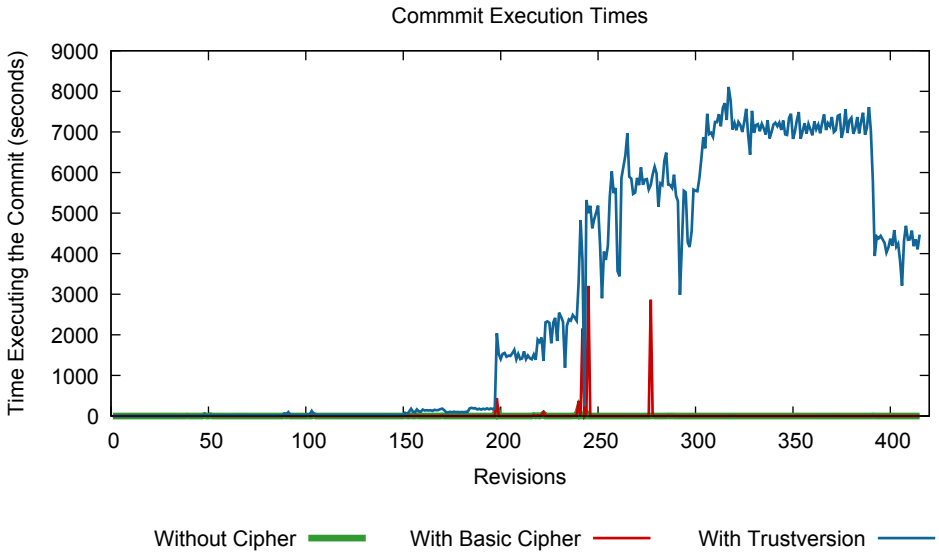


Figure 5.8: Execution time of every commit operation in the CloudStack repository.

The fact that this repository is far larger than all the other repositories we discussed before is important to evaluate the performance overhead introduced by Trustversion, visible in Figure 5.8. We can see that, with the increase of size occupied in the repository, there was also an increase in the execution time per commit performed by append-only encryption. This was in fact affected by the increase in the repository's size, but also greatly affected by the increase in the number of repository files. As we mentioned before, Trustversion does not benefit from Subversion's mechanisms to quickly identify modified files, thus, traversing the entire 4,5GB repository comparing files to their plaintext copies to spot the modified files. This is also the reason why we do not see a significant performance overhead using the basic encryption system, which, in turn, takes advantages of said Subversion mechanisms. These are terrible performance results from Trustversion, once again proving that these kind of repositories are our worst case scenario. The overhead introduced by the append-only encryption is huge, being 1600 times worse than not using encryption and 120 times worse than using the basic encryption Subversion. The highest time to perform a Trustversion commit was approximately 8100 seconds, which translates to almost 2,25 hours. Even the average commit time of 2760 seconds (46 minutes) is an unreasonable amount of time to wait for a commit operation for the amount of space saved in the server.

Nevertheless, we believe that with further development on the append-only encryption system, in particular to employ mechanisms similar to the Subversion's mechanisms to quickly identify which files have not been touched, these performance results could greatly improve.

## Summary

The repository crawler and repository replayer were fundamental for us to gather and recreate big and interesting repositories that truly showed the impact of the 3 systems we compared: the unaltered Subversion system, the Subversion with support for basic file encryption and an append-only encryption

system. We were able to produce results for 4 big software projects, and each one provided us different perspectives on the performance of these systems. Ultimately, what we can retrieve from these results is that append-only encryption can indeed achieve its main objective of protecting all users' files while guaranteeing minimal overhead in storage space usage in the server when comparing with the simple approach of encrypting all the files before every commit. On the other hand, we also noted that the impact on performance on every commit for the append-only encryptions system is huge. Although, we believe this problem could be greatly reduced by adding some optimizations to its methods for detecting modified files, since we realized that the impact on performance is mostly due to the system fully checking every file for content modifications.

The next chapter finishes this thesis by presenting the conclusions regarding the work developed and also introduces some directions in terms of future work.

# 6 Conclusions and Future Work

## 6.1 Conclusions

Security of user data is a major concern nowadays and that is what motivates this work to explore ways of providing secrecy in an area that has not yet been the focus of research in Information Security. That area is Version Control Systems.

Putting a software project under revision control typically means trusting a remote server with our private data, our software project files. However, we can't always trust a remote server. To address this problem, we propose Trustversion, a modified Subversion client that aims to provide a secrecy protected Version Control System. Our solution keeps all files encrypted in the remote server while avoiding a big increase in the size of the repository and also ensuring compatibility with any regular unmodified Subversion server. This is due to taking into account the current behavior of Subversion to store the encrypted files in an efficient manner.

This system was tested and evaluated with big, well known software projects already under Subversion's revision control in order to gather quantitative measurements of its performance and resource usage overhead in comparison to the unmodified Subversion and a modified Subversion with basic encryption functionality.

We showed that Trustversion can indeed meet our goals and requirements for the solution, introducing significantly less storage space overhead than simply encrypting all files in the repository, without altering the server's implementation.

## 6.2 Future Work

To complement the work developed in this thesis, we would like to improve Trustversion with the following design challenges.

### 6.2.1 Improve data privacy

Trustversion's main focus is to maintain the contents of all files in a repository secret. In spite of that, we will extend it to provide secrecy of all user information in the repository, given that the sensible information can be more than just the contents of the files. The Subversion methods for communicating changes completely reveal the directory tree and file names, as we can see in section 2.2. Besides

that, each commit submitted contains information about the user who committed and a log message written by the user about that revision. Both the log message and the information about the user can be considered sensible. We plan to extend Trustversion's basic design with mechanisms to conceal the directory structure, the pathnames of all directories and files and all logging information inside the repository.

Therefore, we would improve Trustversion to circumvent these problems by hiding all that information from the server as well.

### 6.2.2 Improve key management

Even though our basic approach for managing the keys and multiple user access is sufficient to guarantee secrecy of the data, it can be argued that it will become weaker overtime. If we recall the described solution in section 4.1, we can see that all the files in a repository will be encrypted with the same shared key forever, since the beginning of the repository. This is because changing the shared key even once prevents the system from retrieving the revisions encrypted with the previous key, which would break most of the VCS operations. The new key would also have to be sent to every repository user before they received any revisions encrypted with the new key.

We would develop an extension to the Trustversion basic architecture that implements key renewal and still supports the VCS operations and access by multiple users. One possibility would be to maintain in the repository a list of all the keys used for encryption over time, ciphered with every user's public key. This way, every user could still access any revision as long as the key used is ciphered with the his public key and available in the repository. Although, the total of information kept would be the number of keys used times the number of users in the repository. We could also need additional information about which revision uses which key. It may grow considerably, depending on how frequently the repository key is changed.

We can go further with this possibility by using CryptDB's idea of chaining encryption keys (section 3.2). If we introduce *user groups*, assign a key to each group and join the repository users to a group, we can create a chain of encryption. Each shared repository key would be encrypted with each group's key, and each group's key would be encrypted with the public keys of all of its users. This would reduce the space necessary to store information about the keys.

### 6.2.3 Access control

The solution for key management could also be extended to provide access control of the users. Taking the same approach as the previous extension suggesting the use of encryption key chains and user groups, we could define several groups with different access levels and control the user's access by assigning them to a certain group. For example, one group could be restricted to seeing only the latest revisions (and not the whole history) if the latest repository key was the only being encrypted with that group's key. If the group cannot obtain the previous repository keys, the users of that group cannot decrypt changes from revisions encrypted with those keys.

The main challenge of forbidding access to multiple revisions of a file is that some VCS operations go through the process of reconstructing the file, which requires access to all revisions in the history of that file. For example, in a *checkout* operation, where a new user fetches the entire repository for the first time, it is necessary to reconstruct all the files, so that user cannot be forbidden of accessing certain revisions. Thus, for an access control extension to be implemented, we need to find a solution that covers this issue, otherwise, restricting access to revisions blindly will break some of the VCS operations.

# References


[1] CVS: Concurrent Versions System. http://www.nongnu.org/cvs/, 2006.

[2] Apache Subversion. Enterprise class centralized version control for the masses. https://subversion.apache.org/.

[3] Git. Distributed even if your workflow isn't. http://git-scm.com/.

[4] Mercurial. http://mercurial.selenic.com/.

[5] GitHub. Build software better together. https://github.com/.

[6] BitBucket. Unlimited private code repositories. https://bitbucket.org/.

[7] SourceForge. Find Create and Publish Open Source software for free. http://sourceforge.net/.

[8] Google Code Project Hosting. https://code.google.com/.

[9] CodePlex. Project Hosting for Open Source Software. http://www.codeplex.com/.

[10] SourceForge About. http://sourceforge.net/about.

[11] GitHub Press. https://github.com/about/press.

[12] Nickolai Zeldovich Raluca Ada Popa, Catherine M. S. Redfield and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. *Proceedings of the 23rd SOSP*, pages 85–100, Oct 2011.

[13] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. *Proceedings of the VLDB Endowment*, pages 289–300, Mar 2013.

[14] Shay Artzi, Adam Kiezun, Calvin Newport, and David Schultz. Encrypted keyword search in a distributed storage system. *CSAIL Technical Reports*, Feb 2006.

[15] Raluca Ada Popa, Emily Stark, Jonas Helfer, Steven Valdez, Nickolai Zeldovich, M. Frans Kaashoek, and Hari Balakrishnan. Building web applications on top of encrypted data using mylar. *NSDI'14 Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 157–172, 2014.

[16] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Dupless: Server-aided encryption for deduplicated storage. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 179–194, Berkeley, CA, USA, 2013. USENIX Association.

[17] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP '00, pages 44–, Washington, DC, USA, 2000. IEEE Computer Society.

[18] Caroline Fontaine and Fabien Galand. A survey of homomorphic encryption for nonspecialists. *EURASIP Journal on Information Security, Vol.2007, No.15*, Jan 2007.

[19] Apache CloudStack. https://cloudstack.apache.org/, 2016.

[20] Apache's CouchDB. http://couchdb.apache.org/, 2016.

[21] The Perl Programming Language. https://www.perl.org/, 2016.

[22] Apache Ant. http://ant.apache.org/, 2016.

[23] Apache HTTP Server Project. http://httpd.apache.org/.

[24] GCC: the GNU Compiler Collection. https://gcc.gnu.org/.

[25] Mono: Cross platform open source .NET framework. http://www.mono-project.com/.

[26] The FreeBSD Project. https://www.freebsd.org/.

[27] TortoiseSVN: the coolest interface to (Sub)version control. http://tortoisesvn.net/.

[28] FSFS repositories are now the default (server). http://subversion.apache.org/docs/release-notes/1.2.html#fsfs.

[29] The Berkeley DB based repository back-end has been deprecated. http://subversion.apache.org/docs/release-notes/1.8.html#bdb-deprecated.

[30] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. An empirical study of delta algorithms. In *Proceedings of the SCM-6 Workshop on System Configuration Management*, pages 49–66. Springer-Verlag, 1996.

[31] Subversion Notes: svndiff format. https://svn.apache.org/repos/asf/subversion/trunk/notes/svndiff.

[32] EncFS Encrypted Filesystem. http://www.arg0.net/#!encfs/c1awt.

[33] SSH Filesystem. http://fuse.sourceforge.net/sshfs.html.

[34] Crypto Hook. http://sourceforge.net/projects/cryptohook/, 2013.

[35] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM, Vol. 13 No. 7*, pages 422–426, Jul 1970.

[36] Dropbox. https://www.dropbox.com/.

[37] ready where you are Google Drive: All your files. https://www.google.com/drive/.

[38] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22 Nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, Washington, DC, USA, 2002. IEEE Computer Society.

[39] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Message-locked encryption and secure deduplication. In *EUROCRYPT'13*, pages 296–312, 2013.

[40] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Log-structured File Systems*. Arpaci-Dusseau Books, 2014.

[41] Diff Linux man page. https://linux.die.net/man/1/diff.

[42] Diff Unified Format. http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html.